LÊ MINH HOÀNG



GIẢI THUẬT & LẬP TRÌNH

Bài giảng chuyên đề

Đại học Sư phạm Hà Nội, 1999-2002

Lời cảm ơn

Tôi muốn bày tỏ lòng biết ơn đối với những người thầy đã chỉ dạy tận tình trong những năm tháng đầy khó khăn khi tôi mới bước vào học tin học và lập trình. Sự hiểu biết và lòng nhiệt tình của các thầy không những đã cung cấp cho tôi những kiến thức quý báu mà còn là tấm gương sáng cho tôi noi theo khi tôi đứng trên bục giảng cũng với tư cách là một người thầy.

Cuốn tài liệu này được viết dựa trên những tài liệu thu thập được từ nhiều nguồn khác nhau, bởi công sức của nhiều thế hệ thầy trò đã từng giảng dạy và học tập tại Khối Phổ thông chuyên Toán-Tin, Đại học Sư phạm Hà Nội, còn tôi chỉ là người tổng hợp lại. Qua đây, tôi muốn gửi lời cảm ơn tới các đồng nghiệp đã đọc và đóng góp những ý kiến quí báu, cảm ơn các bạn học sinh - những con người đã trực tiếp làm nên cuốn sách này.

Do thời gian hạn hẹp, một số chuyên đề tuy đã có nhưng chưa kịp chỉnh sửa và đưa vào tài liệu. Bạn đọc có thể tham khảo thêm trong phần tra cứu. Rất mong nhận được những lời nhận xét và góp ý của các bạn để hoàn thiện cuốn sách này.

Tokyo, 28 tháng 4 năm 2003

Lê Minh Hoàng

MỤC LỤC

HÌNH VẼ

CHƯƠNG TRÌNH

PHẦN 1. BÀI TOÁN LIỆT KÊ

Có một số bài toán trên thực tế yêu cầu chỉ rõ: trong một tập các đối tượng cho trước có bao nhiêu đối tượng thoả mãn những điều kiện nhất định. Bài toán đó gọi là **bài toán đếm**.

Trong lớp các bài toán đếm, có những bài toán còn yêu cầu chỉ rõ những cấu hình tìm được thoả mãn điều kiện đã cho là những cấu hình nào. Bài toán yêu cầu đưa ra danh sách các cấu hình có thể có gọi là **bài toán liệt kê**.

Để giải bài toán liệt kê, cần phải xác định được một **thuật toán** để có thể theo đó lần lượt xây dựng được tất cả các cấu hình đang quan tâm. Có nhiều phương pháp liệt kê, nhưng chúng cần phải đáp ứng được hai yêu cầu dưới đây:

- Không được lặp lại một cấu hình
- Không được bỏ sót một cấu hình

Có thể nói rằng, phương pháp liệt kê là phương kế cuối cùng để giải được một số bài toán tổ hợp hiện nay. Khó khăn chính của phương pháp này chính là sự bùng nổ tổ hợp dẫn tới sự đòi hỏi lớn về không gian và thời gian thực hiện chương trình. Tuy nhiên cùng với sự phát triển của máy tính điện tử, bằng phương pháp liệt kê, nhiều bài toán tổ hợp đã tìm thấy lời giải. Qua đó, ta cũng nên biết rằng **chỉ nên dùng phương pháp liệt kê khi không còn một phương pháp nào khác** tìm ra lời giải. Chính những nỗ lực giải quyết các bài toán thực tế không dùng phương pháp liệt kê đã thúc đẩy sự phát triển của nhiều ngành toán học.

§1. NHẮC LẠI MỘT SỐ KIẾN THỨC ĐẠI SỐ TỔ HỢP

Cho S là một tập hữu hạn gồm n phần tử và k là một số tự nhiên.

Gọi X là tập các số nguyên dương từ 1 đến k: X = {1, 2, …, k}

1.1. CHỈNH HỢP LẶP

Mỗi ánh xạ f: X → S. Cho tương ứng với mỗi i ∈ X, một và chỉ một phần tử f(i) ∈ S.

Được gọi là một **chỉnh hợp lặp** chập k của S.

Nhưng do X là tập hữu hạn (k phần tử) nên ánh xạ f có thể xác định qua bảng các giá trị f(1), f(2), …, f(k).

Ví dụ: S = {A, B, C, D, E, F}; k = 3. Một ánh xạ f có thể cho như sau:

i	1	2	3
f(i)	E	C	E

Vậy có thể **đồng nhất f với dãy giá trị (f(1), f(2), …, f(k))** và coi dãy giá trị này cũng là một chỉnh hợp lặp chập k của S. Như ví dụ trên (E, C, E) là một chỉnh hợp lặp chập 3 của S. Dễ dàng chứng minh được kết quả sau bằng quy nạp hoặc bằng phương pháp đánh giá khả năng lựa chọn:

Số chỉnh hợp lặp chập k của tập gồm n phần tử:

$$\overline{A}_n^k = n^k$$

1.2. CHỈNH HỢP KHÔNG LẶP

Khi f là đơn ánh có nghĩa là với ∀i, j ∈ X ta có f(i) = f(j) ⇔ i = j. Nói một cách dễ hiểu, khi dãy giá trị f(1), f(2), …, f(k) gồm các phần tử thuộc S **khác nhau đôi một** thì f được gọi là một **chỉnh hợp không lặp chập k** của S. *Ví dụ một chỉnh hợp không lặp (C, A, E):*

i	1	2	3
f(i)	C	A	E

Số chỉnh hợp không lặp chập k của tập gồm n phần tử:

$$A_n^k = n(n-1)(n-2)...(n-k+1) = \frac{n!}{(n-k)!}$$

1.3. HOÁN VỊ

Khi k = n. Một chỉnh hợp không lặp chập n của S được gọi là một **hoán vị** các phần tử của S.

Ví dụ: một hoán vị: (A, D, C, E, B, F) của S = {A, B, C, D, E, F}

i	1	2	3	4	5	6
f(i)	A	D	C	E	B	F

Để ý rằng khi k = n thì số phần tử của tập X = {1, 2, …, n} đúng bằng số phần tử của S. Do tính chất đôi một khác nhau nên dãy f(1), f(2), …, f(n) sẽ liệt kê được hết các phần tử trong S. Như vậy f là toàn ánh. Mặt khác do giả thiết f là chỉnh hợp không lặp nên f là đơn ánh. Ta có tương ứng 1-1

giữa các phần tử của X và S, do đó f là song ánh. Vậy nên ta có thể định nghĩa một hoán vị của S là một song ánh giữa {1, 2, …, n} và S.

Số hoán vị của tập gồm n phần tử = số chỉnh hợp không lặp chập n:

$$P_n = n!$$

1.4. TỔ HỢP

Một tập con gồm k phần tử của S được gọi là một **tổ hợp chập k** của S.

Lấy một tập con k phần tử của S, xét tất cả k! hoán vị của tập con này. Dễ thấy rằng các hoán vị đó là các chỉnh hợp không lặp chập k của S. Ví dụ lấy tập {A, B, C} là tập con của tập S trong ví dụ trên thì: (A, B, C), (C, A, B), (B, C, A), … là các chỉnh hợp không lặp chập 3 của S. Điều đó tức là khi liệt kê tất cả các chỉnh hợp không lặp chập k thì mỗi tổ hợp chập k sẽ được tính k! lần. Vậy:

Số tổ hợp chập k của tập gồm n phần tử:

$$C_n^k = \frac{A_n^k}{k!} = \frac{n!}{k!(n-k)!}$$

Số tập con của tập n phần tử:

$$C_n^0 + C_n^1 + ... + C_n^n = 2^n$$

§2. PHƯƠNG PHÁP SINH (GENERATION)

Phương pháp sinh có thể áp dụng để giải bài toán liệt kê tổ hợp đặt ra nếu như hai điều kiện sau thoả mãn:

- ***Có thể xác định được một thứ tự trên tập các cấu hình tổ hợp cần liệt kê. Từ đó có thể biết đượccấu hình đầu tiên và cấu hình cuối cùng trong thứ tự đó.***
- ***Xây dựng được thuật toán từ một cấu hình chưa phải cấu hình cuối, sinh ra được cấu hình kế tiếp nó.***

Phương pháp sinh có thể mô tả như sau:

```
<Xây dựng cấu hình đầu tiên>;
repeat
  <Đưa ra cấu hình đang có>;
  <Từ cấu hình đang có sinh ra cấu hình kế tiếp nếu còn>;
until <hết cấu hình>;
```

Thứ tự từ điển

Trên các kiểu dữ liệu đơn giản chuẩn, người ta thường nói tới khái niệm thứ tự. Ví dụ trên kiểu số thì có quan hệ: 1 < 2; 2 < 3; 3 < 10; …, trên kiểu ký tự Char thì cũng có quan hệ 'A' < 'B'; 'C' < 'c'…

Xét quan hệ thứ tự toàn phần "nhỏ hơn hoặc bằng" ký hiệu "≤" trên một tập hợp S, là quan hệ hai ngôi thoả mãn bốn tính chất:

Với $\forall a, b, c \in S$

Tính phổ biến: Hoặc là $a \le b$, hoặc $b \le a$;

Tính phản xạ: $a \le a$

Tính phản đối xứng: Nếu $a \le b$ và $b \le a$ thì bắt buộc $a = b$.

Tính bắc cầu: Nếu có $a \le b$ và $b \le c$ thì $a \le c$.

Trong trường hợp $a \le b$ và $a \ne b$, ta dùng ký hiệu "<" cho gọn, (ta ngầm hiểu các ký hiệu như ≥, >, khỏi phải định nghĩa)

Ví dụ như quan hệ "≤" trên các số nguyên cũng như trên các kiểu vô hướng, liệt kê là quan hệ thứ tự toàn phần.

Trên các dãy hữu hạn, người ta cũng xác định một quan hệ thứ tự:

Xét $a = (a_1, a_2, \ldots, a_n)$ và $b = (b_1, b_2, \ldots, b_n)$; trên các phần tử của $a_1, \ldots, a_n, b_1, \ldots, b_n$ đã có quan hệ thứ tự "≤". Khi đó $a \le b$ nếu như

Hoặc $a_i = b_i$ với $\forall i: 1 \le i \le n$.

Hoặc tồn tại một số nguyên dương k: $1 \le k < n$ để:

$a_1 = b_1$

$a_2 = b_2$

…

$a_{k-1} = b_{k-1}$

$a_k = b_k$

$a_{k+1} < b_{k+1}$

Trong trường hợp này, ta có thể viết a < b.

Thứ tự đó gọi là thứ tự từ điển trên các dãy độ dài n.

Khi độ dài hai dãy a và b không bằng nhau, người ta cũng xác định được thứ tự từ điển. Bằng cách thêm vào cuối dãy a hoặc dãy b những phần tử đặc biệt gọi là phần tử ∅ để độ dài của a và b bằng nhau, và coi những phần tử ∅ này nhỏ hơn tất cả các phần tử khác, ta lại đưa về xác định thứ tự từ điển của hai dãy cùng độ dài. Ví dụ:

(1, 2, 3, 4) < (5, 6)

(a, b, c) < (a, b, c, d)

'calculator' < 'computer'

2.1. SINH CÁC DÃY NHỊ PHÂN ĐỘ DÀI N

Một dãy nhị phân độ dài n là một dãy $x = x_1x_2…x_n$ trong đó $x_i \in \{0, 1\}$ ($\forall i : 1 \le i \le n$).

Dễ thấy: một dãy nhị phân x độ dài n là biểu diễn nhị phân của một giá trị nguyên p(x) nào đó nằm trong đoạn $[0, 2^n - 1]$. Số các dãy nhị phân độ dài n = số các số nguyên $\in [0, 2^n - 1] = 2^n$. Ta sẽ lập chương trình liệt kê các dãy nhị phân theo thứ tự từ điển có nghĩa là sẽ liệt kê lần lượt các dãy nhị phân biểu diễn các số nguyên theo thứ tự 0, 1,…, 2^n-1.

Ví dụ: Khi n = 3, các dãy nhị phân độ dài 3 được liệt kê như sau:

p(x)	0	1	2	3	4	5	6	7
x	000	001	010	011	100	101	110	111

Như vậy dãy đầu tiên sẽ là 00…0 và dãy cuối cùng sẽ là 11…1. Nhận xét rằng nếu dãy $x = (x_1, x_2, …, x_n)$ là dãy đang có và không phải dãy cuối cùng thì dãy kế tiếp sẽ nhận được bằng cách cộng thêm 1 (theo cơ số 2 có nhớ) vào dãy hiện tại.

Ví dụ khi n = 8:

```
Dãy đang có:  10010000      Dãy đang có:  10010111
cộng thêm 1:        + 1     cộng thêm 1:        + 1
              ________                    ________
Dãy mới:      10010001      Dãy mới:      10011000
```

Như vậy kỹ thuật sinh cấu hình kế tiếp từ cấu hình hiện tại có thể mô tả như sau: Xét từ cuối dãy về đầu (xét từ hàng đơn vị lên), gặp số 0 đầu tiên thì thay nó bằng số 1 và đặt tất cả các phần tử phía sau vị trí đó bằng 0.

```
i := n;
while (i > 0) and (xi = 1) do i := i - 1;
if i > 0 then
  begin
```

```
    xi := 1;
    for j := i + 1 to n do xj := 0;
  end;
```

Dữ liệu vào (**Input**): nhập từ file văn bản BSTR.INP chứa số nguyên dương $n \le 30$

Kết quả ra (**Output**): ghi ra file văn bản BSTR.OUT các dãy nhị phân độ dài n.

BSTR.INP	BSTR.OUT
3	000 001 010 011 100 101 110 111

P_1_02_1.PAS * Thuật toán sinh liệt kê các dãy nhị phân độ dài n

```
program Binary_Strings;
const
  InputFile  = 'BSTR.INP';
  OutputFile = 'BSTR.OUT';
  max = 30;
var
  x: array[1..max] of Integer;
  n, i: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n);
  Close(f);
  Assign(f, OutputFile); Rewrite(f);
  FillChar(x, SizeOf(x), 0); {Cấu hình ban đầu x1 = x2 = … = xn := 0}
  repeat {Thuật toán sinh}
    for i := 1 to n do Write(f, x[i]); {In ra cấu hình hiện tại}
    WriteLn(f);
    i := n; {xi là phần tử cuối dãy, lùi dần i cho tới khi gặp số 0 hoặc khi i = 0 thì dừng}
    while (i > 0) and (x[i] = 1) do Dec(i);
    if i > 0 then {Chưa gặp phải cấu hình 11…1}
      begin
        x[i] := 1; {Thay xi bằng số 1}
        FillChar(x[i + 1], (n - i) * SizeOf(x[1]), 0); {Đặt xi + 1 = xi + 2 = … = xn := 0}
      end;
  until i = 0; {Đã  hết cấu hình}
  Close(f);
end.
```

2.2. LIỆT KÊ CÁC TẬP CON K PHẦN TỬ

Ta sẽ lập chương trình liệt kê các tập con k phần tử của tập {1, 2, …, n} theo thứ tự từ điển

Ví dụ: với n = 5, k = 3, ta phải liệt kê đủ 10 tập con:

```
1.{1, 2, 3}  2.{1, 2, 4}  3.{1, 2, 5}  4.{1, 3, 4}  5.{1, 3, 5}
6.{1, 4, 5}  7.{2, 3, 4}  8.{2, 3, 5}  9.{2, 4, 5} 10.{3, 4, 5}
```

Như vậy tập con đầu tiên (cấu hình khởi tạo) là {1, 2, …, k}.

Cấu hình kết thúc là {n - k + 1, n - k + 2, …, n}.

Nhận xét: Ta sẽ in ra tập con bằng cách in ra lần lượt các phần tử của nó theo thứ tự tăng dần. Từ đó, ta có nhận xét nếu $x = \{x_1, x_2, …, x_k\}$ và $x_1 < x_2 < … < x_k$ thì giới hạn trên (giá trị lớn nhất có thể nhận) của x_k là n, của x_{k-1} là n - 1, của x_{k-2} là n - 2…

Cụ thể: **giới hạn trên của x_i = n - k + i;**

Còn tất nhiên, **giới hạn dưới của x_i (giá trị nhỏ nhất x_i có thể nhận) là x_{i-1} + 1**.

Như vậy nếu ta đang có một dãy x đại diện cho một tập con, nếu x là cấu hình kết thúc có nghĩa là tất cả các phần tử trong x đều đã đạt tới giới hạn trên thì quá trình sinh kết thúc, nếu không thì ta phải sinh ra một dãy x mới tăng dần thoả mãn **vừa đủ lớn hơn dãy cũ** theo nghĩa không có một tập con k phần tử nào chen giữa chúng khi sắp thứ tự từ điển.

*Ví dụ: n = 9, k = 6. Cấu hình đang có x = {1, 2, 6, 7, 8, 9}. Các phần tử x_3 đến x_6 đã đạt tới giới hạn trên nên để sinh cấu hình mới ta không thể sinh bằng cách tăng một phần tử trong số các x_6, x_5, x_4, x_3 lên được, ta phải tăng x_2 = 2 lên thành x_2 = 3. Được cấu hình mới là x = {1, 3, 6, 7, 8, 9}. Cấu hình này đã thoả mãn lớn hơn cấu hình trước nhưng chưa thoả mãn tính chất **vừa đủ lớn** muốn vậy ta lại thay x_3, x_4, x_5, x_6 bằng các giới hạn dưới của nó. Tức là:*

- $x_3 := x_2 + 1 = 4$
- $x_4 := x_3 + 1 = 5$
- $x_5 := x_4 + 1 = 6$
- $x_6 := x_5 + 1 = 7$

*Ta được cấu hình mới x = {1, **3**, 4, 5, 6, 7} là cấu hình kế tiếp. Nếu muốn tìm tiếp, ta lại nhận thấy rằng x_6 = 7 chưa đạt giới hạn trên, như vậy chỉ cần tăng x_6 lên 1 là được x = {1, 3, 4, 5, 6, **8**}.*

Vậy kỹ thuật sinh tập con kế tiếp từ tập đã có x có thể xây dựng như sau:

Tìm từ cuối dãy lên đầu cho tới khi gặp một phần tử x_i chưa đạt giới hạn trên n - k + i.

```
i := n;
while (i > 0) and (xi = n - k + i) do i := i - 1;
```

(1, 2, 6, 7, 8, 9);

Nếu tìm thấy:

```
if i > 0 then
```

Tăng x_i đó lên 1.

```
xi := xi + 1;
```

(1, 3, 6, 7, 8, 9)

Đặt tất cả các phần tử phía sau x_i bằng giới hạn dưới:

```
for j := i + 1 to k do xj := xj-1 + 1;
```

(1, 3, 4, 5, 6, 7)

Input: file văn bản SUBSET.INP chứa hai số nguyên dương n, k ($1 \le k \le n \le 30$) cách nhau ít nhất một dấu cách

Output: file văn bản SUBSET.OUT các tập con k phần tử của tập {1, 2, …, n}

SUBSET.INP	SUBSET.OUT
5 3	{1, 2, 3} {1, 2, 4} {1, 2, 5} {1, 3, 4} {1, 3, 5} {1, 4, 5} {2, 3, 4} {2, 3, 5} {2, 4, 5} {3, 4, 5}

P_1_02_2.PAS * Thuật toán sinh liệt kê các tập con k phần tử

```
program Combination;
const
  InputFile  = 'SUBSET.INP';
  OutputFile = 'SUBSET.OUT';
  max = 30;
var
  x: array[1..max] of Integer;
  n, k, i, j: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n, k);
  Close(f);
  Assign(f, OutputFile); Rewrite(f);
  for i := 1 to k do x[i] := i; {x1 := 1; x2 := 2; … ; x3 := k   (Cấu hình khởi tạo)}
  repeat
    {In ra cấu hình hiện tại}
    Write(f, '{');
    for i := 1 to k - 1 do Write(f, x[i], ', ');
    WriteLn(f, x[k], '}');
    {Sinh tiếp}
    i := k; {xi là phần tử cuối dãy, lùi dần i cho tới khi gặp một xi chưa đạt giới hạn trên n - k + i}
    while (i > 0) and (x[i] = n - k + i) do Dec(i);
    if i > 0 then {Nếu chưa lùi đến 0 có nghĩa là chưa phải cấu hình kết thúc}
      begin
        Inc(x[i]); {Tăng xi lên 1, Đặt các phần tử đứng sau xi bằng giới hạn dưới của nó}
        for j := i + 1 to k do x[j] := x[j - 1] + 1;
      end;
  until i = 0; {Lùi đến tận 0 có nghĩa là tất cả các phần tử đã đạt giới hạn trên - hết cấu hình}
  Close(f);
end.
```

2.3. LIỆT KÊ CÁC HOÁN VỊ

Ta sẽ lập chương trình liệt kê các hoán vị của {1, 2, …, n} theo thứ tự từ điển.

Ví dụ với n = 4, ta phải liệt kê đủ 24 hoán vị:

```
 1.1234   2.1243   3.1324   4.1342   5.1423   6.1432
 7.2134   8.2143   9.2314  10.2341  11.2413  12.2431
13.3124  14.3142  15.3214  16.3241  17.3412  18.3421
19.4123  20.4132  21.4213  22.4231  23.4312  24.4321
```

Như vậy hoán vị đầu tiên sẽ là (1, 2, …, n). Hoán vị cuối cùng là (n, n-1, … , 1).

Hoán vị sẽ sinh ra phải lớn hơn hoán vị hiện tại, hơn thế nữa phải là hoán vị vừa đủ lớn hơn hoán vị hiện tại theo nghĩa không thể có một hoán vị nào khác chen giữa chúng khi sắp thứ tự.

Giả sử hoán vị hiện tại là x = (3, 2, <u>6, 5, 4, 1</u>), xét 4 phần tử cuối cùng, ta thấy chúng được xếp giảm dần, điều đó có nghĩa là cho dù ta có hoán vị 4 phần tử này thế nào, ta cũng được một hoán vị bé

hơn hoán vị hiện tại!. Như vậy ta phải xét đến x2 = 2, thay nó bằng một giá trị khác. Ta sẽ thay bằng giá trị nào?, không thể là 1 bởi nếu vậy sẽ được hoán vị nhỏ hơn, không thể là 3 vì đã có x1 = 3 rồi (phần tử sau không được chọn vào những giá trị mà phần tử trước đã chọn). Còn lại các giá trị 4, 5, 6. Vì cần một hoán vị **vừa đủ lớn hơn hiện tại** nên ta chọn $x_2 = 4$. Còn các giá trị (x_3, x_4, x_5, x_6) sẽ lấy trong tập {2, 6, 5, 1}. Cũng vì tính vừa đủ lớn nên ta sẽ tìm biểu diễn nhỏ nhất của 4 số này gán cho x_3, x_4, x_5, x_6 tức là (1, 2, 5, 6). Vậy hoán vị mới sẽ là (3, 4, 1, 2, 5, 6).

(3, ***2***, <u>6, 5, 4, 1</u>) → (3, **4, 1, 2, 5, 6**).

Ta có nhận xét gì qua ví dụ này: Đoạn cuối của hoán vị được xếp giảm dần, số $x_5 = 4$ là số nhỏ nhất trong đoạn cuối giảm dần thoả mãn điều kiện lớn hơn $x_2 = 2$. Nếu đổi chỗ x_5 cho x_2 thì ta sẽ được $x_2 = 4$ và đoạn cuối **vẫn được sắp xếp giảm dần**. Khi đó muốn biểu diễn nhỏ nhất cho các giá trị trong đoạn cuối thì ta chỉ cần đảo ngược đoạn cuối.

Trong trường hợp hoán vị hiện tại là (2, 1, 3, 4) thì hoán vị kế tiếp sẽ là (2, 1, 4, 3). Ta cũng có thể coi hoán vị (2, 1, 3, 4) có đoạn cuối giảm dần, đoạn cuối này chỉ gồm 1 phần tử (4)

Vậy kỹ thuật sinh hoán vị kế tiếp từ hoán vị hiện tại có thể xây dựng như sau:

- *Xác định đoạn cuối giảm dần dài nhất, tìm chỉ số i của phần tử x_i đứng liền trước đoạn cuối đó. Điều này đồng nghĩa với việc tìm từ vị trí sát cuối dãy lên đầu, gặp chỉ số i đầu tiên thỏa mãn $x_i < x_{i+1}$. Nếu toàn dãy đã là giảm dần, thì đó là cấu hình cuối.*

```
i := n - 1;
while (i > 0) and (xi > xi+1) do i := i - 1;
```

- *Trong đoạn cuối giảm dần, tìm phần tử x_k nhỏ nhất thoả mãn điều kiện $x_k > x_i$. Do đoạn cuối giảm dần, điều này thực hiện bằng cách tìm từ cuối dãy lên đầu gặp chỉ số k đầu tiên thoả mãn $x_k > x_i$ (có thể dùng tìm kiếm nhị phân).*

```
k := n;
while xk < xi do k := k - 1;
```

- *Đổi chỗ x_k và x_i, lật ngược thứ tự đoạn cuối giảm dần (từ x_{i+1} đến x_k) trở thành tăng dần.*

Input: file văn bản PERMUTE.INP chứa số nguyên dương n ≤ 12

Output: file văn bản PERMUTE.OUT các hoán vị của dãy (1, 2, …, n)

PERMUTE.INP	PERMUTE.OUT
3	1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1

P_1_02_3.PAS * Thuật toán sinh liệt kê hoán vị

```
program Permutation;
const
  InputFile  = 'PERMUTE.INP';
  OutputFile = 'PERMUTE.OUT';
  max = 12;
var
  n, i, k, a, b: Integer;
  x: array[1..max] of Integer;
  f: Text;
```

```
procedure Swap(var X, Y: Integer); {Thủ tục đảo giá trị hai tham biến X, Y}
var
  Temp: Integer;
begin
  Temp := X; X := Y; Y := Temp;
end;

begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n);
  Close(f);
  Assign(f, OutputFile); Rewrite(f);
  for i := 1 to n do x[i] := i; {Khởi tạo cấu hình đầu:  x1 := 1; x2 := 2; …, xn := n}
  repeat
    for i := 1 to n do Write(f, x[i], ' '); {In ra cấu hình hoán vị hiện tại}
    WriteLn(f);
    i := n - 1;
    while (i > 0) and (x[i] > x[i + 1]) do Dec(i);
    if i > 0 then {Chưa gặp phải hoán vị cuối (n, n-1, … ,1)}
      begin
        k := n; {xk là phần tử cuối dãy}
        while x[k] < x[i] do Dec(k); {Lùi dần k để tìm gặp xk đầu tiên lớn hơn xi }
        Swap(x[k], x[i]); {Đổi chỗ xk và xi}
        a := i + 1; b := n; {Lật ngược đoạn cuối giảm dần, a: đầu đoạn, b: cuối đoạn}
        while a < b do
          begin
            Swap(x[a], x[b]); {Đổi chỗ xa và xb}
            Inc(a); {Tiến a và lùi b, đổi chỗ tiếp cho tới khi a, b chạm nhau}
            Dec(b);
          end;
      end;
  until i = 0; {Toàn dãy là dãy giảm dần - không sinh tiếp được - hết cấu hình}
  Close(f);
end.
```

Bài tập:

Bài 1

Các chương trình trên xử lý không tốt trong trường hợp tầm thường, đó là trường hợp n = 0 đối với chương trình liệt kê dãy nhị phân cũng như trong chương trình liệt kê hoán vị, trường hợp k = 0 đối với chương trình liệt kê tổ hợp, hãy khắc phục điều đó.

Bài 2

Liệt kê các dãy nhị phân độ dài n có thể coi là liệt kê các chỉnh hợp lặp chập n của tập 2 phần tử {0, 1}. Hãy lập chương trình:

Nhập vào hai số n và k, liệt kê các chỉnh hợp lặp chập k của {0, 1, …, n -1}.

Gợi ý: thay hệ cơ số 2 bằng hệ cơ số n.

Bài 3

Hãy liệt kê các dãy nhị phân độ dài n mà trong đó cụm chữ số "01" xuất hiện đúng 2 lần.

Bài 4.

Nhập vào một danh sách n tên người. Liệt kê tất cả các cách chọn ra đúng k người trong số n người đó.

Gợi ý: xây dựng một ánh xạ từ tập {1, 2, …, n} đến tập các tên người. Ví dụ xây dựng một mảng Tên: Tên[1] := 'Nguyễn văn A'; Tên[2] := 'Trần thị B';…. sau đó liệt kê tất cả các tập con k phần tử của tập {1, 2, …, n}. Chỉ có điều khi in tập con, ta không in giá trị số {1, 3, 5} mà thay vào đó sẽ in ra {Tên[1], Tên [3], Tên[5]}. Tức là in ra ảnh của các giá trị tìm được qua ánh xạ

Bài 5

Liệt kê tất cả các tập con của tập {1, 2, …, n}. Có thể dùng phương pháp liệt kê tập con như trên hoặc dùng phương pháp liệt kê tất cả các dãy nhị phân. Mỗi số 1 trong dãy nhị phân tương ứng với một phần tử được chọn trong tập. Ví dụ với tập {**1**, 2, **3**, 4} thì dãy nhị phân 1010 sẽ tương ứng với tập con {1, 3}. Hãy lập chương trình in ra tất cả các tập con của {1, 2, …, n} theo hai phương pháp.

Bài 6

Nhập vào danh sách tên n người, in ra tất cả các cách xếp n người đó vào một bàn

Bài 7

Nhập vào danh sách n bạn nam và n bạn nữ, in ra tất cả các cách xếp 2n người đó vào một bàn tròn, mỗi bạn nam tiếp đến một bạn nữ.

Bài 8

Người ta có thể dùng phương pháp sinh để liệt kê các chỉnh hợp không lặp chập k. Tuy nhiên có một cách khác là liệt kê tất cả các tập con k phần tử của tập hợp, sau đó in ra đủ k! hoán vị của nó. Hãy viết chương trình liệt kê các chỉnh hợp không lặp chập k của {1, 2, …, n} theo cả hai cách.

Bài 9

Liệt kê tất cả các hoán vị chữ cái trong từ MISSISSIPPI theo thứ tự từ điển.

Bài 10

Liệt kê tất cả các cách phân tích số nguyên dương n thành tổng các số nguyên dương, hai cách phân tích là hoán vị của nhau chỉ tính là một cách.

Cuối cùng, ta có nhận xét, mỗi phương pháp liệt kê đều có ưu, nhược điểm riêng và phương pháp sinh cũng không nằm ngoài nhận xét đó. Phương pháp sinh **không thể sinh ra được cấu hình thứ p** nếu như chưa có cấu hình thứ p - 1, chứng tỏ rằng phương pháp sinh tỏ ra ưu điểm trong trường hợp liệt kê toàn bộ một **số lượng nhỏ cấu hình trong một bộ dữ liệu lớn** thì lại có nhược điểm và ít tính phổ dụng trong những thuật toán **duyệt hạn chế.** Hơn thế nữa, không phải cấu hình ban đầu lúc nào cũng dễ tìm được, không phải kỹ thuật sinh cấu hình kế tiếp cho mọi bài toán đều đơn giản như trên (Sinh các chỉnh hợp không lặp chập k theo thứ tự từ điển chẳng hạn). Ta sang một chuyên mục sau nói đến một phương pháp liệt kê có tính phổ dụng cao hơn, để giải các bài toán liệt kê phức tạp hơn đó là: Thuật toán quay lui (Back tracking).

§3. THUẬT TOÁN QUAY LUI

Thuật toán quay lui dùng để giải bài toán liệt kê các cấu hình. Mỗi cấu hình được xây dựng bằng cách xây dựng từng phần tử, mỗi phần tử được chọn bằng cách thử tất cả các khả năng.
Giả thiết cấu hình cần liệt kê có dạng (x_1, x_2,…, x_n). Khi đó thuật toán quay lui thực hiện qua các bước sau:

1) Xét tất cả các giá trị x_1 có thể nhận, thử cho x_1 nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x_1 ta sẽ:

2) Xét tất cả các giá trị x_2 có thể nhận, lại thử cho x_2 nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x_2 lại xét tiếp các khả năng chọn x_3 … cứ tiếp tục như vậy đến bước:

…

n) Xét tất cả các giá trị x_n có thể nhận, thử cho x_n nhận lần lượt các giá trị đó, thông báo cấu hình tìm được (x_1, x_2, …, x_n).

Trên phương diện quy nạp, có thể nói rằng thuật toán quay lui liệt kê các cấu hình n phần tử dạng (x_1, x_2, .., x_n) bằng cách thử cho x_1 nhận lần lượt các giá trị có thể. Với mỗi giá trị thử gán cho x_1 lại liệt kê tiếp cấu hình n - 1 phần tử (x_2, x_3, …, x_n).

Mô hình của thuật toán quay lui có thể mô tả như sau:

```
{Thủ tục này thử cho xi nhận lần lượt các giá trị mà nó có thể nhận}
procedure Try(i: Integer);
begin
  for (mọi giá trị V có thể gán cho xi) do
    begin
      <Thử cho xi := V>;
      if (xi là phần tử cuối cùng trong cấu hình) then
           <Thông báo cấu hình tìm được>
      else
        begin
          <Ghi nhận việc cho xi nhận giá trị V (Nếu cần)>;
          Try(i + 1); {Gọi đệ quy để chọn tiếp xi+1}
          <Nếu cần, bỏ ghi nhận việc thử xi := V, để thử giá trị khác>;
        end;
    end;
end;
```

Thuật toán quay lui sẽ bắt đầu bằng lời gọi Try(1)

3.1. LIỆT KÊ CÁC DÃY NHỊ PHÂN ĐỘ DÀI N

Input/Output với khuôn dạng như trong P_1_02_1.PAS

Biểu diễn dãy nhị phân độ dài N dưới dạng (x_1, x_2, …, x_n). Ta sẽ liệt kê các dãy này bằng cách thử dùng các giá trị {0, 1} gán cho x_i. Với mỗi giá trị thử gán cho x_i lại thử các giá trị có thể gán cho x_{i+1}.Chương trình liệt kê bằng thuật toán quay lui có thể viết:

P_1_03_1.PAS * Thuật toán quay lui liệt kê các dãy nhị phân độ dài n

```
program BinaryStrings;
const
```

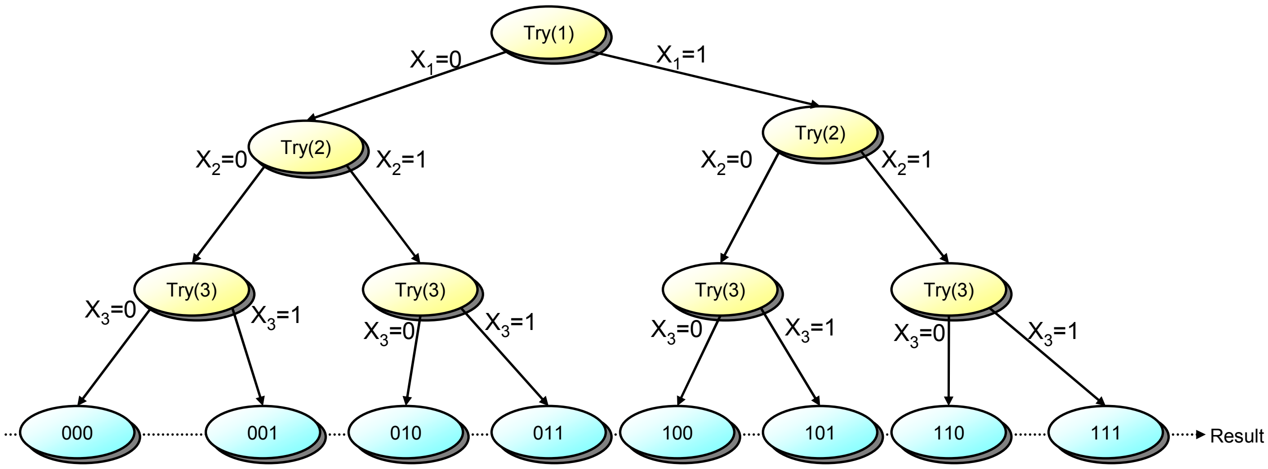
```
  InputFile  = 'BSTR.INP';
  OutputFile = 'BSTR.OUT';
  max = 30;
var
  x: array[1..max] of Integer;
  n: Integer;
  f: Text;

procedure PrintResult; {In cấu hình tìm được, do thủ tục tìm đệ quy Try gọi khi tìm ra một cấu hình}
var
  i: Integer;
begin
  for i := 1 to n do Write(f, x[i]);
  WriteLn(f);
end;

procedure Try(i: Integer); {Thử các cách chọn xi}
var
  j: Integer;
begin
  for j := 0 to 1 do {Xét các giá trị có thể gán cho xi, với mỗi giá trị đó}
    begin
      x[i] := j; {Thử đặt xi}
      if i = n then PrintResult {Nếu i = n thì in kết quả}
      else Try(i + 1); {Nếu i chưa phải là phần tử cuối thì tìm tiếp xi+1}
    end;
end;

begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n); {Nhập dữ liệu}
  Close(f);
  Assign(f, OutputFile); Rewrite(f);
  Try(1); {Thử các cách chọn giá trị x1}
  Close(f);
end.
```

Ví dụ: Khi n = 3, cây tìm kiếm quay lui như sau:



Hình 1: Cây tìm kiếm quay lui trong bài toán liệt kê dãy nhị phân

3.2. LIỆT KÊ CÁC TẬP CON K PHẦN TỬ

Input/Output có khuôn dạng như trong P_1_02_2.PAS

Để liệt kê các tập con k phần tử của tập S = {1, 2, …, n} ta có thể đưa về liệt kê các cấu hình (x_1, x_2, …, x_k) ở đây các $x_i \in S$ và $x_1 < x_2 < \ldots < x_k$. Ta có nhận xét:

$x_k \le n$

$x_{k-1} \le x_k - 1 \le n - 1$

…

$x_i \le n - k + i$

…

$x_1 \le n - k + 1$.

Từ đó suy ra $x_{i-1} + 1 \le x_i \le n - k + i$ ($1 \le i \le k$) ở đây ta giả thiết có thêm một số $x_0 = 0$ khi xét i = 1.

Như vậy ta sẽ xét tất cả các cách chọn x_1 từ 1 (=x_0 + 1) đến n - k + 1, với mỗi giá trị đó, xét tiếp tất cả các cách chọn x_2 từ x_1 + 1 đến n - k + 2,… cứ như vậy khi chọn được đến x_k thì ta có một cấu hình cần liệt kê. Chương trình liệt kê bằng thuật toán quay lui như sau:

P_1_03_2.PAS * Thuật toán quay lui liệt kê các tập con k phần tử

```
program Combination;
const
  InputFile  = 'SUBSET.INP';
  OutputFile = 'SUBSET.OUT';
  max = 30;
var
  x: array[0..max] of Integer;
  n, k: Integer;
  f: Text;

procedure PrintResult;   (*In ra tập con {x1, x2, …, xk}*)
var
  i: Integer;
begin
  Write(f, '{');
  for i := 1 to k - 1 do Write(f, x[i], ', ');
  WriteLn(f, x[k], '}');
end;

procedure Try(i: Integer); {Thử các cách chọn giá trị cho x[i]}
var
  j: Integer;
begin
  for j := x[i - 1] + 1 to n - k + i do
    begin
      x[i] := j;
      if i = k then PrintResult
      else Try(i + 1);
    end;
end;

begin
  Assign(f, InputFile); Reset(F);
  ReadLn(f, n, k);
  Close(f);
  Assign(f, OutputFile); Rewrite(f);
  x[0] := 0;
  Try(1);
  Close(f);
end.
```

Nếu để ý chương trình trên và chương trình liệt kê dãy nhị phân độ dài n, ta thấy về cơ bản chúng chỉ khác nhau ở thủ tục Try(i) - chọn thử các giá trị cho x_i, ở chương trình liệt kê dãy nhị phân ta thử chọn các giá trị 0 hoặc 1 còn ở chương trình liệt kê các tập con k phần tử ta thử chọn x_i là một trong các giá trị nguyên từ x_{i-1} + 1 đến n - k + i. Qua đó ta có thể thấy tính phổ dụng của thuật toán quay lui: mô hình cài đặt có thể thích hợp cho nhiều bài toán, khác với phương pháp sinh tuần tự, với mỗi bài toán lại phải có một thuật toán sinh kế tiếp riêng làm cho việc cài đặt mỗi bài một khác, bên cạnh đó, không phải thuật toán sinh kế tiếp nào cũng dễ cài đặt.

3.3. LIỆT KÊ CÁC CHỈNH HỢP KHÔNG LẶP CHẬP K

Để liệt kê các chỉnh hợp không lặp chập k của tập S = {1, 2, …, n} ta có thể đưa về liệt kê các cấu hình (x_1, x_2, …, x_k) ở đây các $x_i \in S$ và khác nhau đôi một.

Như vậy thủ tục Try(i) - xét tất cả các khả năng chọn x_i - sẽ thử hết các giá trị từ 1 đến n, mà các giá trị này chưa bị các phần tử đứng trước chọn. Muốn xem các giá trị nào chưa được chọn ta sử dụng kỹ thuật dùng mảng đánh dấu:

Khởi tạo một mảng c_1, c_2, …, c_n mang kiểu logic. Ở đây c_i cho biết giá trị i có còn tự do hay đã bị chọn rồi. Ban đầu khởi tạo tất cả các phần tử mảng c là TRUE có nghĩa là các phần tử từ 1 đến n đều tự do.

Tại bước chọn các giá trị có thể của x_i ta chỉ xét những giá trị j có c_j = TRUE có nghĩa là **chỉ chọn những giá trị tự do.**

Trước khi gọi đệ quy tìm xi+1: ta đặt giá trị j vừa gán cho xi là **đã bị chọn** có nghĩa là đặt c_j := FALSE để các thủ tục Try(i + 1), Try(i + 2)… gọi sau này không chọn phải giá trị j đó nữa

Sau khi gọi đệ quy tìm x_{i+1}: có nghĩa là sắp tới ta sẽ thử gán một **giá trị khác** cho xi thì ta sẽ đặt giá trị j vừa thử đó thành **tự do** (cj := TRUE), bởi khi xi đã nhận một giá trị khác rồi thì các phần tử đứng sau: xi+1, xi+2 … hoàn toàn có thể nhận lại giá trị j đó. Điều này hoàn toàn hợp lý trong phép xây dựng chỉnh hợp không lặp: x1 có n cách chọn, x2 có n - 1 cách chọn, …Lưu ý rằng khi thủ tục Try(i) có i = k thì ta không cần phải đánh dấu gì cả vì tiếp theo chỉ có in kết quả chứ không cần phải chọn thêm phần tử nào nữa.

Input: file văn bản ARRANGE.INP chứa hai số nguyên dương n, k (1 ≤ k ≤ n ≤ 20) cách nhau ít nhất một dấu cách

Output: file văn bản ARRANGE.OUT ghi các chỉnh hợp không lặp chập k của tập {1, 2, …, n}

ARRANGE.INP	ARRANGE.OUT
3 2	1 2 1 3 2 1 2 3 3 1 3 2

P_1_03_3.PAS * Thuật toán quay lui liệt kê các chỉnh hợp không lặp chập k

```
program Arrangement;
const
  InputFile  = 'ARRANGES.INP';
  OutputFile = 'ARRANGES.OUT';
  max = 20;
var
  x: array[1..max] of Integer;
  c: array[1..max] of Boolean;
  n, k: Integer;
  f: Text;

procedure PrintResult; {Thủ tục in cấu hình tìm được}
var
  i: Integer;
begin
  for i := 1 to k do Write(f, x[i],' ');
  WriteLn(f);
end;

procedure Try(i: Integer); {Thử các cách chọn xi}
var
  j: Integer;
begin
  for j := 1 to n do
    if c[j] then {Chỉ xét những giá trị j còn tự do}
      begin
        x[i] := j;
        if i = k then PrintResult {Nếu đã chọn được đến xk thì chỉ việc in kết quả}
        else
          begin
            c[j] := False; {Đánh dấu: j đã bị chọn}
            Try(i + 1); {Thủ tục này chỉ xét những giá trị còn tự do gán cho xi+1, tức là sẽ không chọn phải j}
            c[j] := True; {Bỏ đánh dấu: j lại là tự do, bởi sắp tới sẽ thử một cách chọn khác của xi}
          end;
      end;
end;

begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n, k);
  Assign(f, OutputFile); Rewrite(f);
  FillChar(c, SizeOf(c), True); {Tất cả các số đều chưa bị chọn}
  Try(1); {Thử các cách chọn giá trị của x1}
  Close(f);
end.
```

Nhận xét: khi k = n thì đây là chương trình liệt kê hoán vị

3.4. BÀI TOÁN PHÂN TÍCH SỐ

3.4.1. Bài toán

Cho một số nguyên dương $n \le 30$, hãy tìm tất cả các cách phân tích số n thành tổng của các số nguyên dương, các cách phân tích là hoán vị của nhau chỉ tính là 1 cách.

3.4.2. Cách làm:

Ta sẽ lưu nghiệm trong mảng x, ngoài ra có một mảng t. Mảng t xây dựng như sau: t_i sẽ là tổng các phần tử trong mảng x từ x_1 đến x_i: $t_i := x_1 + x_2 + \ldots + x_i$.

Khi liệt kê các dãy x có tổng các phần tử đúng bằng n, để tránh sự trùng lặp ta đưa thêm ràng buộc $x_{i-1} \le x_i$.

Vì số phần tử thực sự của mảng x là không cố định nên thủ tục PrintResult dùng để in ra 1 cách phân tích phải có thêm tham số cho biết sẽ in ra bao nhiêu phần tử.

Thủ tục đệ quy Try(i) sẽ thử các giá trị có thể nhận của x_i ($x_i \ge x_{i-1}$)

Khi nào thì in kết quả và khi nào thì gọi đệ quy tìm tiếp ?

Lưu ý rằng t_{i-1} là tổng của tất cả các phần tử từ x_1 đến x_{i-1} do đó

Khi $t_i = n$ tức là ($x_i = n - t_{i-1}$) thì in kết quả

Khi tìm tiếp, x_{i+1} sẽ phải lớn hơn hoặc bằng x_i. Mặt khác t_{i+1} là tổng của các số từ x_1 tới x_{i+1} không được vượt quá n. Vậy ta có $t_{i+1} \le n \Leftrightarrow t_{i-1} + x_i + x_{i+1} \le n \Leftrightarrow x_i + x_{i+1} \le n - t_{i-1}$ tức là $x_i \le (n - t_{i-1})/2$.

Ví dụ đơn giản khi n = 10 thì chọn x_1 = 6, 7, 8, 9 là việc làm vô nghĩa vì như vậy cũng không ra nghiệm mà cũng không chọn tiếp x_2 được nữa.

Một cách dễ hiểu ta gọi đệ quy tìm tiếp khi giá trị x_i được chọn còn cho phép chọn thêm một phần tử khác lớn hơn hoặc bằng nó mà không làm tổng vượt quá n. Còn ta in kết quả chỉ khi x_i mang giá trị đúng bằng số thiếu hụt của tổng i-1 phần tử đầu so với n.

Vậy thủ tục Try(i) thử các giá trị cho x_i có thể mô tả như sau: (để tổng quát cho i = 1, ta đặt $x_0 = 1$ và $t_0 = 0$).

Xét các giá trị của x_i từ x_{i-1} đến $(n - t_{i-1})$ div 2, cập nhật $t_i := t_{i-1} + x_i$ và gọi đệ quy tìm tiếp.

Cuối cùng xét giá trị $x_i = n - t_{i-1}$ và in kết quả từ x_1 đến x_i.

Input: file văn bản ANALYSE.INP chứa số nguyên dương n ≤ 30

Output: file văn bản ANALYSE.OUT ghi các cách phân tích số n.

ANALYSE.INP	ANALYSE.OUT
6	6 = 1+1+1+1+1+1 6 = 1+1+1+1+2 6 = 1+1+1+3 6 = 1+1+2+2 6 = 1+1+4 6 = 1+2+3 6 = 1+5 6 = 2+2+2 6 = 2+4 6 = 3+3 6 = 6

P_1_03_4.PAS * Thuật toán quay lui liệt kê các cách phân tích số

```
program Analyses;
const
  InputFile  = 'ANALYSE.INP';
  OutputFile = 'ANALYSE.OUT';
  max = 30;
var
  n: Integer;
  x: array[0..max] of Integer;
  t: array[0..max] of Integer;
  f: Text;
```

```
procedure Init; {Khởi tạo}
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n);
  Close(f);
  x[0] := 1;
  t[0] := 0;
end;

procedure PrintResult(k: Integer);
var
  i: Integer;
begin
  Write(f, n, ' = ');
  for i := 1 to k - 1 do Write(f, x[i], '+');
  WriteLn(f, x[k]);
end;

procedure Try(i: Integer);
var
  j: Integer;
begin
  for j := x[i - 1] to (n - T[i - 1]) div 2 do {Trường hợp còn chọn tiếp xi+1}
    begin
      x[i] := j;
      t[i] := t[i - 1] + j;
      Try(i + 1);
    end;
  x[i] := n - T[i - 1]; {Nếu xi là phần tử cuối thì nó bắt buộc phải là … và in kết quả}
  PrintResult(i);
end;

begin
  Init;
  Assign(f, OutputFile); Rewrite(f);
  Try(1);
  Close(f);
end.
```

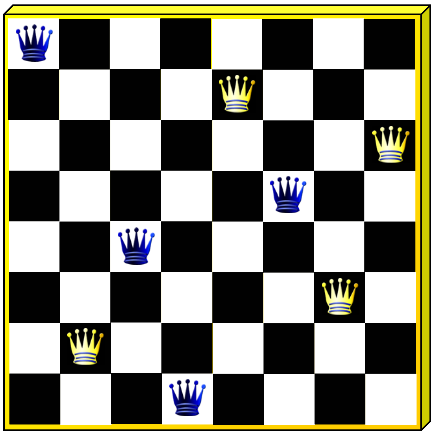
Bây giờ ta xét tiếp một ví dụ kinh điển của thuật toán quay lui:

3.5. BÀI TOÁN XẾP HẬU

3.5.1. Bài toán

Xét bàn cờ tổng quát kích thước nxn. Một quân hậu trên bàn cờ có thể ăn được các quân khác nằm tại các ô cùng hàng, cùng cột hoặc cùng đường chéo. Hãy tìm các xếp n quân hậu trên bàn cờ sao cho không quân nào ăn quân nào.

Ví dụ một cách xếp với n = 8:



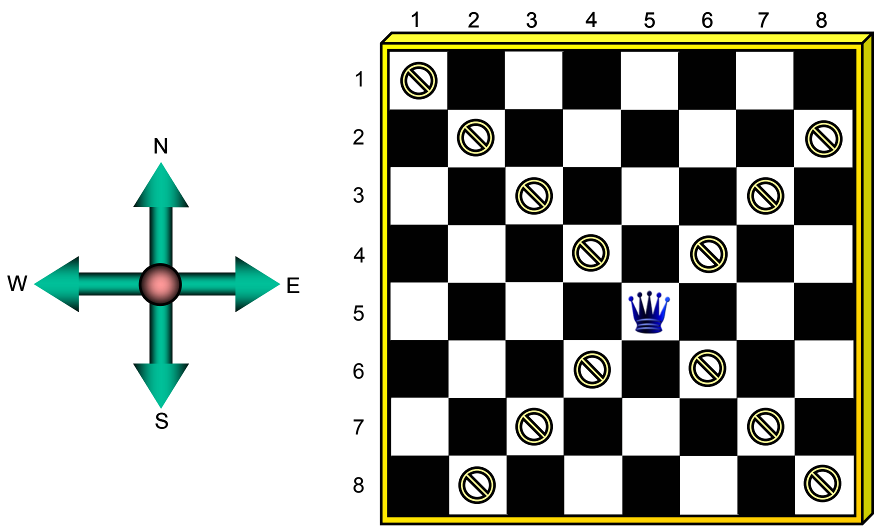
Hình 2: Xếp 8 quân hậu trên bàn cờ 8x8

3.5.2. Phân tích

Rõ ràng n quân hậu sẽ được đặt mỗi con một hàng vì hậu ăn được ngang, ta gọi quân hậu sẽ đặt ở hàng 1 là quân hậu 1, quân hậu ở hàng 2 là quân hậu 2… quân hậu ở hàng n là quân hậu n. Vậy một nghiệm của bài toán sẽ được biết khi ta tìm ra được **vị trí cột của những quân hậu**.

Nếu ta định hướng Đông (Phải), Tây (Trái), Nam (Dưới), Bắc (Trên) thì ta nhận thấy rằng:

- Một đường chéo theo hướng Đông Bắc - Tây Nam (ĐB-TN) bất kỳ sẽ đi qua một số ô, các ô đó có tính chất: Hàng + Cột = C (Const). Với mỗi đường chéo ĐB-TN ta có 1 hằng số C và với một hằng số C: $2 \le C \le 2n$ xác định duy nhất 1 đường chéo ĐB-TN vì vậy ta có thể đánh chỉ số cho các đường chéo ĐB- TN từ 2 đến 2n
- Một đường chéo theo hướng Đông Nam - Tây Bắc (ĐN-TB) bất kỳ sẽ đi qua một số ô, các ô đó có tính chất: Hàng - Cột = C (Const). Với mỗi đường chéo ĐN-TB ta có 1 hằng số C và với một hằng số C: $1 - n \le C \le n - 1$ xác định duy nhất 1 đường chéo ĐN-TB vì vậy ta có thể đánh chỉ số cho các đường chéo ĐN- TB từ 1 - n đến n - 1.



Hình 3: Đường chéo ĐB-TN mang chỉ số 10 và đường chéo ĐN-TB mang chỉ số 0

Cài đặt:

Ta có 3 mảng logic để đánh dấu:

- Mảng a[1..n]. a_i = TRUE nếu như cột i còn tự do, a_i = FALSE nếu như cột i đã bị một quân hậu khống chế
- Mảng b[2..2n]. b_i = TRUE nếu như đường chéo ĐB-TN thứ i còn tự do, b_i = FALSE nếu như đường chéo đó đã bị một quân hậu khống chế.
- Mảng c[1 - n..n - 1]. c_i = TRUE nếu như đường chéo ĐN-TB thứ i còn tự do, c_i = FALSE nếu như đường chéo đó đã bị một quân hậu khống chế.

Ban đầu cả 3 mảng đánh dấu đều mang giá trị TRUE. (Các cột và đường chéo đều tự do)

Thuật toán quay lui:

- Xét tất cả các cột, thử đặt quân hậu 1 vào một cột, với mỗi cách đặt như vậy, xét tất cả các cách đặt quân hậu 2 không bị quân hậu 1 ăn, lại thử 1 cách đặt và xét tiếp các cách đặt quân hậu 3…Mỗi cách đặt được đến quân hậu n cho ta 1 nghiệm
- Khi chọn vị trí cột j cho quân hậu thứ i, thì ta phải chọn ô(i, j) không bị các quân hậu đặt trước đó ăn, tức là phải chọn cột j còn tự do, đường chéo ĐB-TN (i+j) còn tự do, đường chéo ĐN-TB(i-j) còn tự do. Điều này có thể kiểm tra ($a_j = b_{i+j} = c_{i-j}$ = TRUE)
- Khi thử đặt được quân hậu thứ i vào cột j, nếu đó là quân hậu cuối cùng (i = n) thì ta có một nghiệm. Nếu không:
    - **Trước khi gọi** đệ quy tìm cách đặt quân hậu thứ i + 1, ta đánh dấu cột và 2 đường chéo bị quân hậu vừa đặt khống chế ($a_j = b_{i+j} = c_{i-j}$:= FALSE) để các lần gọi đệ quy tiếp sau chọn cách đặt các quân hậu kế tiếp sẽ không chọn vào những ô nằm trên cột j và những đường chéo này nữa.
    - **Sau khi gọi** đệ quy tìm cách đặt quân hậu thứ i + 1, có nghĩa là sắp tới ta lại thử một cách đặt khác cho quân hậu thứ i, ta bỏ đánh dấu cột và 2 đường chéo bị quân hậu vừa thử đặt khống chế ($a_j = b_{i+j} = c_{i-j}$:= TRUE) tức là cột và 2 đường chéo đó lại thành tự do, bởi khi đã đặt quân hậu i sang vị trí khác rồi thì cột và 2 đường chéo đó hoàn toàn có thể gán cho một quân hậu khác

Hãy xem lại trong các chương trình liệt kê chỉnh hợp không lặp và hoán vị về kỹ thuật đánh dấu. Ở đây chỉ khác với liệt kê hoán vị là: liệt kê hoán vị chỉ cần một mảng đánh dấu xem giá trị có tự do không, còn bài toán xếp hậu thì cần phải đánh dấu cả 3 thành phần: Cột, đường chéo ĐB-TN, đường chéo ĐN- TB. Trường hợp đơn giản hơn: Yêu cầu liệt kê các cách đặt n quân xe lên bàn cờ nxn sao cho không quân nào ăn quân nào chính là bài toán liệt kê hoán vị

- **Input:** file văn bản QUEENS.INP chứa số nguyên dương n ≤ 12
- **Output:** file văn bản QUEENS.OUT, mỗi dòng ghi một cách đặt n quân hậu

QUEENS.INP	QUEENS.OUT
5	(1, 1); (2, 3); (3, 5); (4, 2); (5, 4); (1, 1); (2, 4); (3, 2); (4, 5); (5, 3); (1, 2); (2, 4); (3, 1); (4, 3); (5, 5); (1, 2); (2, 5); (3, 3); (4, 1); (5, 4); (1, 3); (2, 1); (3, 4); (4, 2); (5, 5); (1, 3); (2, 5); (3, 2); (4, 4); (5, 1); (1, 4); (2, 1); (3, 3); (4, 5); (5, 2); (1, 4); (2, 2); (3, 5); (4, 3); (5, 1); (1, 5); (2, 2); (3, 4); (4, 1); (5, 3); (1, 5); (2, 3); (3, 1); (4, 4); (5, 2);

P_1_03_5.PAS * Thuật toán quay lui giải bài toán xếp hậu

```
program n_Queens;
const
  InputFile  = 'QUEENS.INP';
  OutputFile = 'QUEENS.OUT';
  max = 12;
var
  n: Integer;
  x: array[1..max] of Integer;
  a: array[1..max] of Boolean;
  b: array[2..2 * max] of Boolean;
  c: array[1 - max..max - 1] of Boolean;
  f: Text;

procedure Init;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n);
  Close(f);
  FillChar(a, SizeOf(a), True); {Mọi cột đều tự do}
  FillChar(b, SizeOf(b), True); {Mọi đường chéo Đông Bắc - Tây Nam đều tự do}
  FillChar(c, SizeOf(c), True); {Mọi đường chéo Đông Nam - Tây Bắc đều tự do}
end;

procedure PrintResult;
var
  i: Integer;
begin
  for i := 1 to n do Write(f, '(', i, ', ', x[i], '); ');
  WriteLn(f);
end;

procedure Try(i: Integer); {Thử các cách đặt quân hậu thứ i vào hàng i}
var
  j: Integer;
begin
  for j := 1 to n do
    if a[j] and b[i + j] and c[i - j] then {Chỉ xét những cột j mà ô (i, j) chưa bị khống chế}
      begin
        x[i] := j; {Thử đặt quân hậu i vào cột j}
        if i = n then PrintResult
        else
          begin
            a[j] := False; b[i + j] := False; c[i - j] := False; {Đánh dấu}
            Try(i + 1); {Tìm các cách đặt quân hậu thứ i + 1}
            a[j] := True; b[i + j] := True; c[i - j] := True; {Bỏ đánh dấu}
          end;
      end;
end;

begin
```

```
  Init;
  Assign(f, OutputFile); Rewrite(f);
  Try(1);
  Close(f);
end.
```

Tên gọi thuật toán quay lui, đứng trên phương diện cài đặt có thể nên gọi là kỹ thuật vét cạn bằng quay lui thì chính xác hơn, tuy nhiên đứng trên phương diện bài toán, nếu như ta coi công việc giải bài toán bằng cách xét tất cả các khả năng cũng là 1 cách giải thì tên gọi Thuật toán quay lui cũng không có gì trái logic. Xét hoạt động của chương trình trên cây tìm kiếm quay lui ta thấy tại bước thử chọn x_i nó sẽ gọi đệ quy để tìm tiếp x_{i+1} có nghĩa là quá trình sẽ duyệt tiến sâu xuống phía dưới đến tận nút lá, sau khi đã duyệt hết các nhánh, tiến trình lùi lại thử áp đặt một giá trị khác cho x_i, đó chính là nguồn gốc của tên gọi "thuật toán quay lui"

Bài tập:

Bài 1

Một số chương trình trên xử lý không tốt trong trường hợp tầm thường (n = 0 hoặc k = 0), hãy khắc phục các lỗi đó

Bài 2

Viết chương trình liệt kê các chỉnh hợp lặp chập k của n phần tử

Bài 3

Cho hai số nguyên dương l, n. Hãy liệt kê các xâu nhị phân độ dài n có tính chất, bất kỳ hai xâu con nào độ dài l liền nhau đều khác nhau.

Bài 4

Với n = 5, k = 3, vẽ cây tìm kiếm quay lui của chương trình liệt kê tổ hợp chập k của tập {1, 2, …, n}

Bài 5

Liệt kê tất cả các tập con của tập S gồm n số nguyên $\{S_1, S_2, \ldots, S_n\}$ nhập vào từ bàn phím

Bài 6

Tương tự như bài 5 nhưng chỉ liệt kê các tập con có max - min ≤ T (T cho trước).

Bài 7

Một dãy $(x_1, x_2, \ldots, x_n)$ gọi là một hoán vị hoàn toàn của tập {1, 2,…, n} nếu nó là một hoán vị và thoả mãn $x_i \neq i$ với $\forall i$: $1 \leq i \leq n$. Hãy viết chương trình liệt kê tất cả các hoán vị hoàn toàn của tập trên (n vào từ bàn phím).

Bài 8

Sửa lại thủ tục in kết quả (PrintResult) trong bài xếp hậu để có thể vẽ hình bàn cờ và các cách đặt hậu ra màn hình.

Bài 9

Mã đi tuần: Cho bàn cờ tổng quát kích thước nxn và một quân Mã, hãy chỉ ra một hành trình của quân Mã xuất phát từ ô đang đứng đi qua tất cả các ô còn lại của bàn cờ, mỗi ô đúng 1 lần.

Bài 10

Chuyển tất cả các bài tập trong bài trước đang viết bằng sinh tuần tự sang quay lui.

Bài 11

Xét sơ đồ giao thông gồm n nút giao thông đánh số từ 1 tới n và m đoạn đường nối chúng, mỗi đoạn đường nối 2 nút giao thông. Hãy nhập dữ liệu về mạng lưới giao thông đó, nhập số hiệu hai nút giao thông s và d. Hãy in ra tất cả các cách đi từ s tới d mà mỗi cách đi không được qua nút giao thông nào quá một lần.

§4. KỸ THUẬT NHÁNH CẬN

4.1. BÀI TOÁN TỐI ƯU

Một trong những bài toán đặt ra trong thực tế là việc tìm ra **một** nghiệm thoả mãn một số điều kiện nào đó, và nghiệm đó là **tốt nhất** theo một chỉ tiêu cụ thể, nghiên cứu lời giải các lớp bài toán tối ưu thuộc về lĩnh vực quy hoạch toán học. Tuy nhiên cũng cần phải nói rằng trong nhiều trường hợp chúng ta chưa thể xây dựng một thuật toán nào thực sự hữu hiệu để giải bài toán, mà cho tới nay việc tìm nghiệm của chúng vẫn phải dựa trên mô hình **liệt kê** toàn bộ các cấu hình có thể và đánh giá, tìm ra cấu hình tốt nhất. Việc liệt kê cấu hình có thể cài đặt bằng các phương pháp liệt kê: Sinh tuần tự và tìm kiếm quay lui. Dưới đây ta sẽ tìm hiểu phương pháp liệt kê bằng thuật toán quay lui để tìm nghiệm của bài toán tối ưu.

4.2. SỰ BÙNG NỔ TỔ HỢP

Mô hình thuật toán quay lui là tìm kiếm trên 1 cây phân cấp. Nếu giả thiết rằng ứng với mỗi nút tương ứng với một giá trị được chọn cho x_i sẽ ứng với chỉ 2 nút tương ứng với 2 giá trị mà x_{i+1} có thể nhận thì cây n cấp sẽ có tới 2^n nút lá, con số này lớn hơn rất nhiều lần so với dữ liệu đầu vào n. Chính vì vậy mà nếu như ta có thao tác thừa trong việc chọn x_i thì sẽ phải trả giá rất lớn về chi phí thực thi thuật toán bởi quá trình tìm kiếm lòng vòng vô nghĩa trong các bước chọn kế tiếp x_{i+1}, x_{i+2}, … Khi đó, một vấn đề đặt ra là trong quá trình liệt kê lời giải ta cần tận dụng những thông tin đã tìm được để loại bỏ sớm những phương án chắc chắn không phải tối ưu. Kỹ thuật đó gọi là kỹ thuật đánh giá nhánh cận trong tiến trình quay lui.

4.3. MÔ HÌNH KỸ THUẬT NHÁNH CẬN

Dựa trên mô hình thuật toán quay lui, ta xây dựng mô hình sau:

```
procedure Init;
begin
  <Khởi tạo một cấu hình bất kỳ BESTCONFIG>;
end;

{Thủ tục này thử chọn cho xi tất cả các giá trị nó có thể nhận}
procedure Try(i: Integer);
begin
  for (Mọi giá trị V có thể gán cho xi) do
    begin
      <Thử cho xi := V>;
      if (Việc thử trên vẫn còn hi vọng tìm ra cấu hình tốt hơn BESTCONFIG) then
        if (xi là phần tử cuối cùng trong cấu hình) then
          <Cập nhật BESTCONFIG>
        else
          begin
            <Ghi nhận việc thử xi = V nếu cần>;
            Try(i + 1); {Gọi đệ quy, chọn tiếp xi+1 }
            <Bỏ ghi nhận việc thử cho xi = V (nếu cần)>;
          end;
```

```
    end;
end;

begin
   Init;
   Try(1);
   <Thông báo cấu hình tối ưu BESTCONFIG>
end.
```

Kỹ thuật nhánh cận thêm vào cho thuật toán quay lui khả năng đánh giá theo từng bước, nếu tại bước thứ i, giá trị thử gán cho x_i không có hi vọng tìm thấy cấu hình tốt hơn cấu hình BESTCONFIG thì thử giá trị khác ngay mà không cần phải gọi đệ quy tìm tiếp hay ghi nhận kết quả làm gì. Nghiệm của bài toán sẽ được làm tốt dần, bởi khi tìm ra một cấu hình mới (tốt hơn BESTCONFIG - tất nhiên), ta không in kết quả ngay mà sẽ cập nhật BESTCONFIG bằng cấu hình mới vừa tìm được

4.4. BÀI TOÁN NGƯỜI DU LỊCH

4.4.1. Bài toán

Cho n thành phố đánh số từ 1 đến n và m tuyến đường giao thông hai chiều giữa chúng, mạng lưới giao thông này được cho bởi bảng C cấp nxn, ở đây $C_{ij} = C_{ji}$ = Chi phí đi đoạn đường trực tiếp từ thành phố i đến thành phố j. Giả thiết rằng $C_{ii} = 0$ với $\forall i$, $C_{ij} = +\infty$ nếu không có đường trực tiếp từ thành phố i đến thành phố j.

Một người du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại mỗi thành phố đúng 1 lần và cuối cùng quay lại thành phố 1. Hãy chỉ ra cho người đó hành trình với chi phí ít nhất. Bài toán đó gọi là bài toán người du lịch hay bài toán hành trình của một thương gia (Traveling Salesman)

4.4.2. Cách giải

Hành trình cần tìm có dạng ($x_1 = 1, x_2, \ldots, x_n, x_{n+1} = 1$) ở đây giữa x_i và x_{i+1}: hai thành phố liên tiếp trong hành trình phải có đường đi trực tiếp ($C_{ij} \neq +\infty$) và ngoại trừ thành phố 1, không thành phố nào được lặp lại hai lần. Có nghĩa là dãy ($x_1, x_2, \ldots, x_n$) lập thành 1 hoán vị của (1, 2, …, n).

Duyệt quay lui: x_2 có thể chọn một trong các thành phố mà x_1 có đường đi tới (trực tiếp), với mỗi cách thử chọn x_2 như vậy thì x_3 có thể chọn một trong các thành phố mà x_2 có đường đi tới (ngoài x_1). Tổng quát: x_i có thể chọn 1 trong các thành phố **chưa đi qua** mà **từ x_{i-1} có đường đi trực tiếp tới ($1 \le i \le n$).**

Nhánh cận: Khởi tạo cấu hình BestConfig có chi phí = $+\infty$. Với mỗi bước thử chọn x_i xem chi phí đường đi cho tới lúc đó có < Chi phí của cấu hình BestConfig?, nếu không nhỏ hơn thì thử giá trị khác ngay bởi có đi tiếp cũng chỉ tốn thêm. Khi thử được một giá trị x_n ta kiểm tra xem x_n có đường đi trực tiếp về 1 không ? Nếu có đánh giá chi phí đi từ thành phố 1 đến thành phố x_n cộng với chi

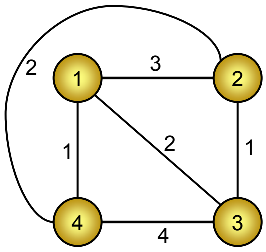
phí từ x_n đi trực tiếp về 1, nếu nhỏ hơn chi phí của đường đi BestConfig thì cập nhật lại BestConfig bằng cách đi mới.

Sau thủ tục tìm kiếm quay lui mà chi phí của BestConfig vẫn bằng +∞ thì có nghĩa là nó không tìm thấy một hành trình nào thoả mãn điều kiện đề bài để cập nhật BestConfig, bài toán không có lời giải, còn nếu chi phí của BestConfig < +∞ thì in ra cấu hình BestConfig - đó là hành trình ít tốn kém nhất tìm được

Input: file văn bản TOURISM.INP

- Dòng 1: Chứa số thành phố n (1 ≤ n ≤ 20) và số tuyến đường m trong mạng lưới giao thông
- m dòng tiếp theo, mỗi dòng ghi số hiệu hai thành phố có đường đi trực tiếp và chi phí đi trên quãng đường đó (chi phí này là số nguyên dương ≤ 100)

Output: file văn bản TOURISM.OUT, ghi hành trình tìm được.



TOURISM.INP	TOURISM.OUT
4 6 1 2 3 1 3 2 1 4 1 2 3 1 2 4 2 3 4 4	1->3->2->4->1 Cost: 6

P_1_04_1.PAS * Kỹ thuật nhánh cận dùng cho bài toán người du lịch

```
program TravellingSalesman;
const
  InputFile  = 'TOURISM.INP';
  OutputFile = 'TOURISM.OUT';
  max = 20;
  maxC = 20 * 100 + 1;{+∞}
var
  C: array[1..max, 1..max] of Integer; {Ma trận chi phí}
  X, BestWay: array[1..max + 1] of Integer; {X để thử các khả năng, BestWay để ghi nhận nghiệm}
  T: array[1..max + 1] of Integer; {Ti để lưu chi phí đi từ X1 đến Xi}
  Free: array[1..max] of Boolean; {Free để đánh dấu, Freei= True nếu chưa đi qua tp i}
  m, n: Integer;
  MinSpending: Integer; {Chi phí hành trình tối ưu}

procedure Enter;
var
  i, j, k: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n, m);
  for i := 1 to n do {Khởi tạo bảng chi phí ban đầu}
    for j := 1 to n do
      if i = j then C[i, j] := 0 else C[i, j] := maxC;
  for k := 1 to m do
    begin
      ReadLn(f, i, j, C[i, j]);
      C[j, i] := C[i, j]; {Chi phí như nhau trên 2 chiều}
    end;
  Close(f);
end;

procedure Init; {Khởi tạo}
```

```
begin
  FillChar(Free, n, True);
  Free[1] := False; {Các thành phố là chưa đi qua ngoại trừ thành phố 1}
  X[1] := 1; {Xuất phát từ thành phố 1}
  T[1] := 0; {Chi phí tại thành phố xuất phát là 0}
  MinSpending := maxC;
end;

procedure Try(i: Integer); {Thử các cách chọn xi}
var
  j: Integer;
begin
  for j := 2 to n do {Thử các thành phố từ 2 đến n}
    if Free[j] then {Nếu gặp thành phố chưa đi qua}
      begin
        X[i] := j; {Thử đi}
        T[i] := T[i - 1] + C[x[i - 1], j]; {Chi phí := Chi phí bước trước + chi phí đường đi trực tiếp}
        if T[i] < MinSpending then {Hiển nhiên nếu có điều này thì C[x[i - 1], j] < +∞ rồi}
          if i < n then {Nếu chưa đến được xn}
            begin
              Free[j] := False; {Đánh dấu thành phố vừa thử}
              Try(i + 1); {Tìm các khả năng chọn xi+1}
              Free[j] := True; {Bỏ đánh dấu}
            end
          else
            if T[n] + C[x[n], 1] < MinSpending then {Từ xn quay lại 1 vẫn tốn chi phí ít hơn trước}
              begin {Cập nhật BestConfig}
                BestWay := X;
                MinSpending := T[n] + C[x[n], 1];
              end;
      end;
end;

procedure PrintResult;
var
  i: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  if MinSpending = maxC then WriteLn(f, 'NO SOLUTION')
  else
    for i := 1 to n do Write(f, BestWay[i], '->');
  WriteLn(f, 1);
  WriteLn(f, 'Cost: ', MinSpending);
  Close(f);
end;

begin
  Enter;
  Init;
  Try(2);
  PrintResult;
end.
```

Trên đây là một giải pháp nhánh cận còn rất thô sơ giải bài toán người du lịch, trên thực tế người ta còn có nhiều cách đánh giá nhánh cận chặt hơn nữa. Hãy tham khảo các tài liệu khác để tìm hiểu về những phương pháp đó.

4.5. DÃY ABC

Cho trước một số nguyên dương N (N ≤ 100), hãy tìm một xâu chỉ gồm các ký tự A, B, C thoả mãn 3 điều kiện:

Có độ dài N

Hai đoạn con bất kỳ liền nhau đều khác nhau (đoạn con là một dãy ký tự liên tiếp của xâu)

Có ít ký tự C nhất.

Cách giải:

Không trình bày, đề nghị tự xem chương trình để hiểu, chỉ chú thích kỹ thuật nhánh cận như sau:

Nếu dãy $X_1X_2…X_n$ thoả mãn 2 đoạn con bất kỳ liền nhau đều khác nhau, thì trong 4 ký tự liên tiếp bất kỳ bao giờ cũng phải có 1 ký tự "C". Như vậy với một dãy con gồm k ký tự liên tiếp của dãy X thì số ký tự C trong dãy con đó bắt buộc phải ≥ k div 4.

Tại bước thử chọn X_i, nếu ta đã có T_i ký tự "C" trong đoạn đã chọn từ X_1 đến X_i, thì cho dù các bước đệ quy tiếp sau làm tốt như thế nào chăng nữa, số ký tự "C" sẽ phải chọn thêm bao giờ cũng ≥ (n - i) div 4. Tức là nếu theo phương án chọn X_i như thế này thì số ký tự "C" trong dãy kết quả (khi chọn đến X_n) cho dù có làm tốt đến đâu cũng ≥ $\mathbf{T_i}$ **+ (n - i) div 4**. Ta dùng con số này để đánh giá nhánh cận, nếu nó nhiều hơn số ký tự "C" trong BestConfig thì chắc chắn có làm tiếp cũng chỉ được một cấu hình tồi tệ hơn, ta bỏ qua ngay cách chọn này và thử phương án khác.

Input: file văn bản ABC.INP chứa số nguyên dương n ≤ 100

Output: file văn bản ABC.OUT ghi xâu tìm được

ABC.INP	ABC.OUT
10	ABACABCBAB "C" Letter Count : 2

P_1_04_2.PAS * Dãy ABC

```
program ABC_STRING;
const
  InputFile  = 'ABC.INP';
  OutputFile = 'ABC.OUT';
  max = 100;
var
  N, MinC: Integer;
  X, Best: array[1..max] of 'A'..'C';
  T: array[0..max] of Integer; {Ti cho biết số ký tự "C" trong đoạn từ X1 đến Xi}
  f: Text;

{Hàm Same(i, l) cho biết xâu gồm l ký tự kết thúc tại Xi có trùng với xâu l ký tự liền trước nó không ?}
function Same(i, l: Integer): Boolean;
var
  j, k: Integer;
begin
  j := i - l; {j là vị trí cuối đoạn liền trước đoạn đó}
  for k := 0 to l - 1 do
    if X[i - k] <> X[j - k] then
      begin
        Same := False; Exit;
      end;
  Same := True;
```

```
end;

{Hàm Check(i) cho biết Xi có làm hỏng tính không lặp của dãy X1X2 … Xi hay không}
function Check(i: Integer): Boolean;
var
  l: Integer;
begin
  for l := 1 to i div 2 do {Thử các độ dài l}
    if Same(i, l) then {Nếu có xâu độ dài l kết thúc bởi Xi  bị trùng với xâu liền trước}
      begin
        Check := False; Exit;
      end;
  Check := True;
end;

{Giữ lại kết quả vừa tìm được vào BestConfig (MinC và mảng Best)}
procedure KeepResult;
begin
  MinC := T[N];
  Best := X;
end;

{Thuật toán quay lui có nhánh cận}
procedure Try(i: Integer); {Thử các giá trị có thể của Xi}
var
  j: 'A'..'C';
begin
  for j := 'A' to 'C' do {Xét tất cả các giá trị}
    begin
      X[i] := j;
      if Check(i) then {Nếu thêm giá trị đó vào không làm hỏng tính không lặp }
        begin
          if j = 'C' then T[i] := T[i - 1] + 1 {Tính Ti  qua Ti - 1}
          else T[i] := T[i - 1];
          if T[i] + (N - i) div 4 < MinC then {Đánh giá nhánh cận}
            if i = N then KeepResult
            else Try(i + 1);
        end;
    end;
end;

procedure PrintResult;
var
  i: Integer;
begin
  for i := 1 to N do Write(f, Best[i]);
  WriteLn(f);
  WriteLn(f, '"C" Letter Count  : ', MinC);
end;

begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, N);
  Close(f);
  Assign(f, OutputFile); Rewrite(f);
  T[0] := 0;
  MinC := N; {Khởi tạo cấu hình BestConfig ban đầu rất tồi}
  Try(1);
  PrintResult;
  Close(f);
end.
```

Nếu ta thay bài toán là tìm xâu ít ký tự 'B' nhất mà vẫn viết chương trình tương tự như trên thì chương trình sẽ chạy chậm hơn chút ít. Lý do: thủ tục Try ở trên sẽ thử lần lượt các giá trị 'A', 'B',

rồi mới đến 'C'. Có nghĩa ngay trong cách tìm, nó đã tiết kiệm sử dụng ký tự 'C' nhất nên trong phần lớn các bộ dữ liệu nó nhanh chóng tìm ra lời giải hơn so với bài toán tương ứng tìm xâu ít ký tự 'B' nhất. Chính vì vậy mà nếu như đề bài yêu cầu ít ký tự 'B' nhất ta cứ lập chương trình làm yêu cầu ít ký tự 'C' nhất, chỉ có điều khi in kết quả, ta đổi vai trò 'B', 'C' cho nhau. Đây là một ví dụ cho thấy sức mạnh của thuật toán quay lui khi kết hợp với kỹ thuật nhánh cận, nếu viết quay lui thuần tuý hoặc đánh giá nhánh cận không tốt thì với N = 100, tôi cũng không đủ kiên nhẫn để đợi chương trình cho kết quả (chỉ biết rằng > 3 giờ). Trong khi đó khi N = 100, với chương trình trên chỉ chạy hết hơn 3 giây cho kết quả là xâu 27 ký tự 'C'.

Nói chung, ít khi ta gặp bài toán mà chỉ cần sử dụng một thuật toán, một mô hình kỹ thuật cài đặt là có thể giải được. Thông thường các bài toán thực tế đòi hỏi phải có sự tổng hợp, pha trộn nhiều thuật toán, nhiều kỹ thuật mới có được một lời giải tốt. Không được lạm dụng một kỹ thuật nào và cũng không xem thường một phương pháp nào khi bắt tay vào giải một bài toán tin học. Thuật toán quay lui cũng không phải là ngoại lệ, ta phải biết phối hợp một cách uyển chuyển với các thuật toán khác thì khi đó nó mới thực sự là một công cụ mạnh.

Bài tập:

Bài 1

Một dãy dấu ngoặc hợp lệ là một dãy các ký tự "(" và ")" được định nghĩa như sau:

i. Dãy rỗng là một dãy dấu ngoặc hợp lệ độ sâu 0

ii. Nếu A là dãy dấu ngoặc hợp lệ độ sâu k thì (A) là dãy dấu ngoặc hợp lệ độ sâu k + 1

iii. Nếu A và B là hay dãy dấu ngoặc hợp lệ với độ sâu lần lượt là p và q thì AB là dãy dấu ngoặc hợp lệ độ sâu là max(p, q)

Độ dài của một dãy ngoặc là tổng số ký tự "(" và ")"

Ví dụ: Có 5 dãy dấu ngoặc hợp lệ độ dài 8 và độ sâu 3:

```
1. ((()()))
2. ((())())
3. ((()))()
4. (()(()))
5. ()((()))
```

Bài toán đặt ra là khi cho biết trước hai số nguyên dương n và k. Hãy liệt kê hết các dãy ngoặc hợp lệ có độ dài là n và độ sâu là k (làm được với n càng lớn càng tốt).

Bài 2

Cho một bãi mìn kích thước mxn ô vuông, trên một ô có thể có chứa một quả mìn hoặc không, để biểu diễn bản đồ mìn đó, người ta có hai cách:

Cách 1: dùng bản đồ đánh dấu: sử dụng một lưới ô vuông kích thước mxn, trên đó tại ô (i, j) ghi số 1 nếu ô đó có mìn, ghi số 0 nếu ô đó không có mìn

Cách 2: dùng bản đồ mật độ: sử dụng một lưới ô vuông kích thước mxn, trên đó tại ô (i, j) ghi một số trong khoảng từ 0 đến 8 cho biết tổng số mìn trong các ô lân cận với ô (i, j) (ô lân cận với ô (i, j) là ô có chung với ô (i, j) ít nhất 1 đỉnh).

Giả thiết rằng hai bản đồ được ghi chính xác theo tình trạng mìn trên hiện trường.

Về nguyên tắc, lúc cài bãi mìn phải vẽ cả bản đồ đánh dấu và bản đồ mật độ, tuy nhiên sau một thời gian dài, khi người ta muốn gỡ mìn ra khỏi bãi thì vấn đề hết sức khó khăn bởi bản đồ đánh dấu đã bị thất lạc !!. ***Công việc của các lập trình viên là: Từ bản đồ mật độ, hãy tái tạo lại bản đồ đánh dấu của bãi mìn.***

Dữ liệu: Vào từ file văn bản MINE.INP, các số trên 1 dòng cách nhau ít nhất 1 dấu cách

- Dòng 1: Ghi 2 số nguyên dương m, n (2 ≤ m, n ≤ 30)
- m dòng tiếp theo, dòng thứ i ghi n số trên hàng i của bản đồ mật độ theo đúng thứ tự từ trái qua phải.

Kết quả: Ghi ra file văn bản MINE.OUT, các số trên 1 dòng ghi cách nhau ít nhất 1 dấu cách

- Dòng 1: Ghi tổng số lượng mìn trong bãi
- m dòng tiếp theo, dòng thứ i ghi n số trên hàng i của bản đồ đánh dấu theo đúng thứ tự từ trái qua phải.

Ví dụ:

MINE.INP	MINE.OUT
10 15	80
0 3 2 3 3 3 5 3 4 4 5 4 4 4 3	1 0 1 1 1 1 0 1 1 1 1 1 1 1 1
1 4 3 5 5 4 5 4 7 7 7 5 6 6 5	0 0 1 0 0 1 1 1 0 1 1 1 0 1 1
1 4 3 5 4 3 5 4 4 4 4 3 4 5 5	0 0 1 0 0 1 0 0 1 1 1 0 0 1 1
1 4 2 4 4 5 4 2 4 4 3 2 3 5 4	1 0 1 1 1 0 0 1 0 0 0 0 0 1 1
1 3 2 5 4 4 2 2 3 2 3 3 2 5 2	1 0 0 0 1 1 1 0 0 1 0 0 1 0 1
2 3 2 3 3 5 3 2 4 4 3 4 2 4 1	0 0 0 0 1 0 0 0 0 1 1 0 1 0 0
2 3 2 4 3 3 2 3 4 6 6 5 3 3 1	0 1 1 0 0 1 0 0 1 1 0 0 1 0 0
2 6 4 5 2 4 1 3 3 5 5 5 6 4 3	1 0 1 0 1 0 1 0 1 1 1 1 0 1 0
4 6 5 7 3 5 3 5 5 6 5 4 4 4 3	0 1 1 0 1 0 0 0 0 0 1 1 1 1 1
2 4 4 4 2 3 1 2 2 2 3 3 3 4 2	1 1 1 1 1 0 1 1 1 1 0 0 0 0 1

PHẦN 2. CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Hạt nhân của các chương trình máy tính là sự lưu trữ và xử lý thông tin. Việc tổ chức dữ liệu như thế nào có ảnh hưởng rất lớn đến cách thức xử lý dữ liệu đó cũng như tốc độ thực thi và sự chiếm dụng bộ nhớ của chương trình. Việc đặc tả bằng các cấu trúc tổng quát (generic structures) và các kiểu dữ liệu trừu tượng (abstract data types) còn cho phép người lập trình có thể dễ dàng hình dung ra các công việc cụ thể và giảm bớt công sức trong việc chỉnh sửa, nâng cấp và sử dụng lại các thiết kế đã có.

Mục đích của phần này là cung cấp những hiểu biết nền tảng trong việc thiết kế một chương trình máy tính, để thấy rõ được sự cần thiết của việc phân tích, lựa chọn cấu trúc dữ liệu phù hợp cho từng bài toán cụ thể; đồng thời khảo sát một số cấu trúc dữ liệu và thuật toán kinh điển mà lập trình viên nào cũng cần phải nắm vững.

§1. CÁC BƯỚC CƠ BẢN KHI TIẾN HÀNH GIẢI CÁC BÀI TOÁN TIN HỌC

1.1. XÁC ĐỊNH BÀI TOÁN

Input → Process → Output

(Dữ liệu vào → Xử lý → Kết quả ra)

Việc xác định bài toán tức là phải xác định xem ta phải giải quyết vấn đề gì?, với giả thiết nào đã cho và lời giải cần phải đạt những yêu cầu gì. Khác với bài toán thuần tuý toán học chỉ cần xác định rõ giả thiết và kết luận chứ không cần xác định yêu cầu về lời giải, đôi khi những bài toán tin học ứng dụng trong thực tế chỉ cần tìm lời giải tốt tới mức nào đó, thậm chí là tồi ở mức chấp nhận được. Bởi lời giải tốt nhất đòi hỏi quá nhiều thời gian và chi phí.

Ví dụ:

Khi cài đặt các hàm số phức tạp trên máy tính. Nếu tính bằng cách khai triển chuỗi vô hạn thì độ chính xác cao hơn nhưng thời gian chậm hơn hàng tỉ lần so với phương pháp xấp xỉ. Trên thực tế việc tính toán luôn luôn cho phép chấp nhận một sai số nào đó nên các hàm số trong máy tính đều được tính bằng phương pháp xấp xỉ của giải tích số

Xác định đúng yêu cầu bài toán là rất quan trọng bởi nó ảnh hưởng tới cách thức giải quyết và chất lượng của lời giải. Một bài toán thực tế thường cho bởi những thông tin khá mơ hồ và hình thức, ta phải phát biểu lại một cách chính xác và chặt chẽ để hiểu đúng bài toán.

Ví dụ:

- *Bài toán: Một dự án có n người tham gia thảo luận, họ muốn chia thành các nhóm và mỗi nhóm thảo luận riêng về một phần của dự án. Nhóm có bao nhiêu người thì được trình lên bấy nhiêu ý kiến. Nếu lấy ở mỗi nhóm một ý kiến đem ghép lại thì được một bộ ý kiến triển khai dự án. Hãy tìm cách chia để số bộ ý kiến cuối cùng thu được là lớn nhất.*
- *Phát biểu lại: Cho một số nguyên dương n, tìm các phân tích n thành tổng các số nguyên dương sao cho tích của các số đó là lớn nhất.*

Trên thực tế, ta nên xét một vài trường hợp cụ thể để thông qua đó hiểu được bài toán rõ hơn và thấy được các thao tác cần phải tiến hành. Đối với những bài toán đơn giản, đôi khi chỉ cần qua ví dụ là ta đã có thể đưa về một bài toán quen thuộc để giải.

1.2. TÌM CẤU TRÚC DỮ LIỆU BIỂU DIỄN BÀI TOÁN

Khi giải một bài toán, ta cần phải định nghĩa tập hợp dữ liệu để biểu diễn tình trạng cụ thể. Việc lựa chọn này tuỳ thuộc vào vấn đề cần giải quyết và những thao tác sẽ tiến hành trên dữ liệu vào. Có những thuật toán chỉ thích ứng với một cách tổ chức dữ liệu nhất định, đối với

những cách tổ chức dữ liệu khác thì sẽ kém hiệu quả hoặc không thể thực hiện được. Chính vì vậy nên bước xây dựng cấu trúc dữ liệu không thể tách rời bước tìm kiếm thuật toán giải quyết vấn đề.

Các tiêu chuẩn khi lựa chọn cấu trúc dữ liệu

Cấu trúc dữ liệu trước hết phải biểu diễn được đầy đủ các thông tin nhập và xuất của bài toán

Cấu trúc dữ liệu phải phù hợp với các thao tác của thuật toán mà ta lựa chọn để giải quyết bài toán.

Cấu trúc dữ liệu phải cài đặt được trên máy tính với ngôn ngữ lập trình đang sử dụng

Đối với một số bài toán, trước khi tổ chức dữ liệu ta phải viết một đoạn chương trình nhỏ để **khảo sát** xem dữ liệu cần lưu trữ lớn tới mức độ nào.

1.3. TÌM THUẬT TOÁN

Thuật toán là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy thao tác trên cấu trúc dữ liệu sao cho: Với một bộ dữ liệu vào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta đạt được mục tiêu đã định.

Các đặc trưng của thuật toán

1.3.1. Tính đơn định

Ở mỗi bước của thuật toán, các thao tác phải hết sức rõ ràng, không gây nên sự nhập nhằng, lộn xộn, tuỳ tiện, đa nghĩa. Thực hiện đúng các bước của thuật toán thì với một dữ liệu vào, chỉ cho duy nhất một kết quả ra.

1.3.2. Tính dừng

Thuật toán không được rơi vào quá trình vô hạn, phải dừng lại và cho kết quả sau một số hữu hạn bước.

1.3.3. Tính đúng

Sau khi thực hiện tất cả các bước của thuật toán theo đúng quá trình đã định, ta phải được kết quả mong muốn với mọi bộ dữ liệu đầu vào. Kết quả đó được kiểm chứng bằng yêu cầu bài toán.

1.3.4. Tính phổ dụng

Thuật toán phải dễ sửa đổi để thích ứng được với bất kỳ bài toán nào trong một lớp các bài toán và có thể làm việc trên các dữ liệu khác nhau.

1.3.5. Tính khả thi

a) Kích thước phải đủ nhỏ: Ví dụ: Một thuật toán sẽ có tính hiệu quả bằng 0 nếu lượng bộ nhớ mà nó yêu cầu vượt quá khả năng lưu trữ của hệ thống máy tính.

b) Thuật toán phải được máy tính thực hiện trong thời gian cho phép, điều này khác với lời giải toán (Chỉ cần chứng minh là kết thúc sau hữu hạn bước). Ví dụ như xếp thời khoá biểu cho một học kỳ thì không thể cho máy tính chạy tới học kỳ sau mới ra được.

c) Phải dễ hiểu và dễ cài đặt.

Ví dụ:

Input: 2 số nguyên tự nhiên a và b không đồng thời bằng 0

Output: Ước số chung lớn nhất của a và b

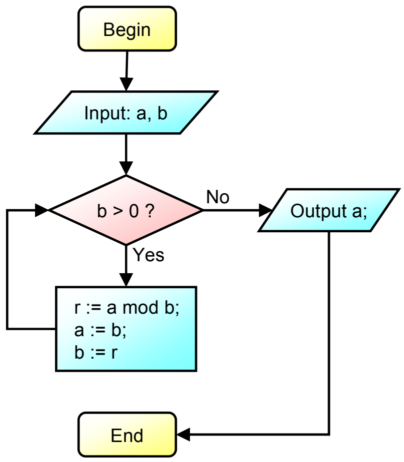
Thuật toán sẽ tiến hành được mô tả như sau: (Thuật toán Euclide)

Bước 1 (Input): Nhập a và b: Số tự nhiên

Bước 2: Nếu b $\neq 0$ thì chuyển sang bước 3, nếu không thì bỏ qua bước 3, đi làm bước 4

Bước 3: Đặt r := a mod b; Đặt a := b; Đặt b := r; Quay trở lại bước 2.

Bước 4 (Output): Kết luận ước số chung lớn nhất phải tìm là giá trị của a. Kết thúc thuật toán.



Hình 4: Lưu đồ thuật giải (Flowchart)

Khi mô tả thuật toán bằng ngôn ngữ tự nhiên, ta không cần phải quá chi tiết các bước và tiến trình thực hiện mà chỉ cần mô tả một cách hình thức đủ để chuyển thành ngôn ngữ lập trình. Viết sơ đồ các thuật toán đệ quy là một ví dụ.

Đối với những thuật toán phức tạp và nặng về tính toán, các bước và các công thức nên mô tả một cách tường minh và chú thích rõ ràng để khi lập trình ta có thể nhanh chóng tra cứu.

Đối với những thuật toán kinh điển thì phải thuộc. Khi giải một bài toán lớn trong một thời gian giới hạn, ta chỉ phải thiết kế tổng thể còn những chỗ đã thuộc thì cứ việc lắp ráp vào. Tính đúng đắn của những mô-đun đã thuộc ta không cần phải quan tâm nữa mà tập trung giải quyết các phần khác.

1.4. LẬP TRÌNH

Sau khi đã có thuật toán, ta phải tiến hành lập trình thể hiện thuật toán đó. Muốn lập trình đạt hiệu quả cao, cần phải có kỹ thuật lập trình tốt. Kỹ thuật lập trình tốt thể hiện ở kỹ năng viết chương trình, khả năng gỡ rối và thao tác nhanh. Lập trình tốt không phải chỉ cần nắm vững ngôn ngữ lập trình là đủ, phải biết cách viết chương trình uyển chuyển, khôn khéo và phát triển dần dần để chuyển các ý tưởng ra thành chương trình hoàn chỉnh. Kinh nghiệm cho thấy một thuật toán hay nhưng do cài đặt vụng về nên khi chạy lại cho kết quả sai hoặc tốc độ chậm.

Thông thường, ta không nên cụ thể hoá ngay toàn bộ chương trình mà nên tiến hành theo phương pháp tinh chế từng bước (Stepwise refinement):

Ban đầu, chương trình được thể hiện bằng ngôn ngữ tự nhiên, thể hiện thuật toán với các bước tổng thể, mỗi bước nêu lên một công việc phải thực hiện.

Một công việc đơn giản hoặc là một đoạn chương trình đã được học thuộc thì ta tiến hành viết mã lệnh ngay bằng ngôn ngữ lập trình.

Một công việc phức tạp thì ta lại chia ra thành những công việc nhỏ hơn để lại tiếp tục với những công việc nhỏ hơn đó.

Trong quá trình tinh chế từng bước, ta phải đưa ra những biểu diễn dữ liệu. Như vậy cùng với sự tinh chế các công việc, dữ liệu cũng được tinh chế dần, có cấu trúc hơn, thể hiện rõ hơn mối liên hệ giữa các dữ liệu.

Phương pháp tinh chế từng bước là một thể hiện của tư duy giải quyết vấn đề từ trên xuống, giúp cho người lập trình có được một định hướng thể hiện trong phong cách viết chương trình. Tránh việc mò mẫm, xoá đi viết lại nhiều lần, biến chương trình thành tờ giấy nháp.

1.5. KIỂM THỬ

1.5.1. Chạy thử và tìm lỗi

Chương trình là do con người viết ra, mà đã là con người thì ai cũng có thể nhầm lẫn. Một chương trình viết xong chưa chắc đã chạy được ngay trên máy tính để cho ra kết quả mong muốn. Kỹ năng tìm lỗi, sửa lỗi, điều chỉnh lại chương trình cũng là một kỹ năng quan trọng của người lập trình. Kỹ năng này chỉ có được bằng kinh nghiệm tìm và sửa chữa lỗi của chính mình.

Có ba loại lỗi:

Lỗi cú pháp: Lỗi này hay gặp nhất nhưng lại dễ sửa nhất, chỉ cần nắm vững ngôn ngữ lập trình là đủ. Một người được coi là không biết lập trình nếu không biết sửa lỗi cú pháp.

Lỗi cài đặt: Việc cài đặt thể hiện không đúng thuật toán đã định, đối với lỗi này thì phải xem lại tổng thể chương trình, kết hợp với các chức năng gỡ rối để sửa lại cho đúng.

Lỗi thuật toán: Lỗi này ít gặp nhất nhưng nguy hiểm nhất, nếu nhẹ thì phải điều chỉnh lại thuật toán, nếu nặng thì có khi phải loại bỏ hoàn toàn thuật toán sai và làm lại từ đầu.

1.5.2. Xây dựng các bộ test

Có nhiều chương trình rất khó kiểm tra tính đúng đắn. Nhất là khi ta không biết kết quả đúng là thế nào?. Vì vậy nếu như chương trình vẫn chạy ra kết quả (không biết đúng sai thế nào) thì việc tìm lỗi rất khó khăn. Khi đó ta nên làm các bộ test để thử chương trình của mình.

Các bộ test nên đặt trong các file văn bản, bởi việc tạo một file văn bản rất nhanh và mỗi lần chạy thử chỉ cần thay tên file dữ liệu vào là xong, không cần gõ lại bộ test từ bàn phím. Kinh nghiệm làm các bộ test là:

Bắt đầu với một bộ test nhỏ, đơn giản, làm bằng tay cũng có được đáp số để so sánh với kết quả chương trình chạy ra.

Tiếp theo vẫn là các bộ test nhỏ, nhưng chứa các giá trị đặc biệt hoặc tầm thường. Kinh nghiệm cho thấy đây là những test dễ sai nhất.

Các bộ test phải đa dạng, tránh sự lặp đi lặp lại các bộ test tương tự.

Có một vài test lớn chỉ để kiểm tra tính chịu đựng của chương trình mà thôi. Kết quả có đúng hay không thì trong đa số trường hợp, ta không thể kiểm chứng được với test này.

Lưu ý rằng chương trình chạy qua được hết các test không có nghĩa là chương trình đó đã đúng. Bởi có thể ta chưa xây dựng được bộ test làm cho chương trình chạy sai. Vì vậy nếu có thể, ta nên tìm cách chứng minh tính đúng đắn của thuật toán và chương trình, điều này thường rất khó.

1.6. TỐI ƯU CHƯƠNG TRÌNH

Một chương trình đã chạy đúng không có nghĩa là việc lập trình đã xong, ta phải sửa đổi lại một vài chi tiết để chương trình có thể chạy nhanh hơn, hiệu quả hơn. Thông thường, trước khi kiểm thử thì ta nên đặt mục tiêu viết chương trình sao cho đơn giản, **miễn sao chạy ra kết quả đúng** là được, sau đó khi tối ưu chương trình, ta xem lại những chỗ nào viết chưa tốt thì tối ưu lại mã lệnh để chương trình ngắn hơn, chạy nhanh hơn. Không nên viết tới đâu tối ưu mã đến đó, bởi chương trình có mã lệnh tối ưu thường phức tạp và khó kiểm soát.

Việc tối ưu chương trình nên dựa trên các tiêu chuẩn sau:

1.6.1. Tính tin cậy

Chương trình phải chạy đúng như dự định, mô tả đúng một giải thuật đúng. Thông thường khi viết chương trình, ta luôn có thói quen kiểm tra tính đúng đắn của các bước mỗi khi có thể.

1.6.2. Tính uyển chuyển

Chương trình phải dễ sửa đổi. Bởi ít có chương trình nào viết ra đã hoàn hảo ngay được mà vẫn cần phải sửa đổi lại. Chương trình viết dễ sửa đổi sẽ làm giảm bớt công sức của lập trình viên khi phát triển chương trình.

1.6.3. Tính trong sáng

Chương trình viết ra phải dễ đọc dễ hiểu, để sau một thời gian dài, khi đọc lại còn hiểu mình làm cái gì?. Để nếu có điều kiện thì còn có thể sửa sai (nếu phát hiện lỗi mới), cải tiến hay biến đổi để được chương trình giải quyết bài toán khác. Tính trong sáng của chương trình phụ thuộc rất nhiều vào công cụ lập trình và phong cách lập trình.

1.6.4. Tính hữu hiệu

Chương trình phải chạy nhanh và ít tốn bộ nhớ, tức là tiết kiệm được cả về không gian và thời gian. Để có một chương trình hữu hiệu, cần phải có giải thuật tốt và những tiểu xảo khi lập trình. Tuy nhiên, việc áp dụng quá nhiều tiểu xảo có thể khiến chương trình trở nên rối rắm, khó hiểu khi sửa đổi. Tiêu chuẩn hữu hiệu nên dừng lại ở mức chấp nhận được, không quan trọng bằng ba tiêu chuẩn trên. Bởi phần cứng phát triển rất nhanh, yêu cầu hữu hiệu không cần phải đặt ra quá nặng.

Từ những phân tích ở trên, chúng ta nhận thấy rằng việc làm ra một chương trình đòi hỏi rất nhiều công đoạn và tiêu tốn khá nhiều công sức. Chỉ một công đoạn không hợp lý sẽ làm tăng chi phí viết chương trình. Nghĩ ra cách giải quyết vấn đề đã khó, biến ý tưởng đó thành hiện thực cũng không dễ chút nào.

Những cấu trúc dữ liệu và giải thuật đề cập tới trong chuyên đề này là những kiến thức rất phổ thông, một người học lập trình không sớm thì muộn cũng phải biết tới. Chỉ hy vọng rằng khi học xong chuyên đề này, qua những cấu trúc dữ liệu và giải thuật hết sức mẫu mực, chúng ta rút ra được bài học kinh nghiệm: **Đừng bao giờ viết chương trình khi mà chưa suy xét kỹ về giải thuật và những dữ liệu cần thao tác**, bởi như vậy ta dễ mắc phải hai sai lầm trầm trọng: hoặc là sai về giải thuật, hoặc là giải thuật không thể triển khai nổi trên một cấu trúc dữ liệu không phù hợp. Chỉ cần mắc một trong hai lỗi đó thôi thì nguy cơ sụp đổ toàn bộ chương trình là hoàn toàn có thể, càng cố chữa càng bị rối, khả năng hầu như chắc chắn là phải làm lại từ đầu[(*)].

[(*)] Tất nhiên, cẩn thận đến đâu thì cũng có xác suất rủi ro nhất định, ta hiểu được mức độ tai hại của hai lỗi này để hạn chế nó càng nhiều càng tốt

§2. PHÂN TÍCH THỜI GIAN THỰC HIỆN GIẢI THUẬT

2.1. ĐỘ PHỨC TẠP TÍNH TOÁN CỦA GIẢI THUẬT

Với một bài toán không chỉ có một giải thuật. Chọn một giải thuật đưa tới kết quả nhanh nhất là một đòi hỏi thực tế. Như vậy cần có một căn cứ nào đó để nói rằng giải thuật này nhanh hơn giải thuật kia ?.

Thời gian thực hiện một giải thuật bằng chương trình máy tính phụ thuộc vào rất nhiều yếu tố. Một yếu tố cần chú ý nhất đó là kích thước của dữ liệu đưa vào. Dữ liệu càng lớn thì thời gian xử lý càng chậm, chẳng hạn như thời gian sắp xếp một dãy số phải chịu ảnh hưởng của số lượng các số thuộc dãy số đó. Nếu gọi n là kích thước dữ liệu đưa vào thì thời gian thực hiện của một giải thuật có thể biểu diễn một cách tương đối như một hàm của n: T(n).

Phần cứng máy tính, ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy đều ảnh hưởng tới thời gian thực hiện. Những yếu tố này không giống nhau trên các loại máy, vì vậy không thể dựa vào chúng khi xác định T(n). Tức là T(n) không thể biểu diễn bằng đơn vị thời gian giờ, phút, giây được. Tuy nhiên, không phải vì thế mà không thể so sánh được các giải thuật về mặt tốc độ. Nếu như thời gian thực hiện một giải thuật là $T_1(n) = n^2$ và thời gian thực hiện của một giải thuật khác là $T_2(n) = 100n$ thì khi n đủ lớn, thời gian thực hiện của giải thuật T_2 rõ ràng nhanh hơn giải thuật T_1. Khi đó, nếu nói rằng thời gian thực hiện giải thuật tỉ lệ thuận với n hay tỉ lệ thuận với n^2 cũng cho ta một cách đánh giá tương đối về tốc độ thực hiện của giải thuật đó khi n khá lớn. Cách đánh giá thời gian thực hiện giải thuật độc lập với máy tính và các yếu tố liên quan tới máy tính như vậy sẽ dẫn tới khái niệm gọi là **độ phức tạp tính toán của giải thuật**.

Cho f và g là hai hàm xác định dương với mọi n. Hàm f(n) được gọi là O(g(n)) nếu tồn tại một hằng số c > 0 và một giá trị n_0 sao cho:

$$f(n) \le c.g(n) \text{ với } \forall\ n \ge n_0$$

Nghĩa là nếu xét những giá trị $n \ge n_0$ thì hàm f(n) sẽ bị chặn trên bởi một hằng số nhân với g(n). Khi đó, nếu f(n) là thời gian thực hiện của một giải thuật thì ta nói giải thuật đó có cấp là g(n), ký hiệu: O(g(n))[(*)] hay Θ(g(n)).

2.2. XÁC ĐỊNH ĐỘ PHỨC TẠP TÍNH TOÁN CỦA GIẢI THUẬT

Việc xác định độ phức tạp tính toán của một giải thuật bất kỳ có thể rất phức tạp. Tuy nhiên, trong thực tế, đối với một số giải thuật ta có thể phân tích bằng một số quy tắc đơn giản:

(*) Ký pháp O(.) được gọi là ký pháp chữ O lớn (big O notation)

2.2.1. Quy tắc cộng

Nếu đoạn chương trình P_1 có thời gian thực hiện $T_1(n) = O(f(n))$ và đoạn chương trình P_2 có thời gian thực hiện là $T_2(n) = O(g(n))$ thì thời gian thực hiện P_1 rồi đến P_2 tiếp theo sẽ là

$$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$$

Chứng minh:

$T_1(n) = O(f(n))$ nên $\exists$ n_1 và c_1 để $T_1(n) \leq c_1.f(n)$ với $\forall$ $n \geq n_1$.

$T_2(n) = O(g(n))$ nên $\exists$ n_2 và c_2 để $T_2(n) \leq c_2.g(n)$ với $\forall$ $n \geq n_2$.

Chọn $n_0 = \max(n_1, n_2)$ và $c = \max(c_1, c_2)$ ta có:

Với $\forall$ $n \geq n_0$:

$T_1(n) + T_2(n) \leq c_1.f(n) + c_2.g(n) \leq c.f(n) + c.g(n) \leq c.(f(n) + g(n)) \leq 2c.(\max(f(n), g(n)))$.

Vậy $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$.

2.2.2. Quy tắc nhân

Nếu đoạn chương trình P có thời gian thực hiện là $T(n) = O(f(n))$. Khi đó, nếu thực hiện k(n) lần đoạn chương trình P với $k(n) = O(g(n))$ thì độ phức tạp tính toán sẽ là $O(g(n).f(n))$

Chứng minh:

Thời gian thực hiện k(n) lần đoạn chương trình P sẽ là k(n)T(n). Theo định nghĩa:

$\exists$ $c_k \geq 0$ và n_k để $k(n) \leq c_k(g(n))$ với $\forall$ $n \geq n_k$

$\exists$ $c_T \geq 0$ và n_T để $T(n) \leq c_T(f(n))$ với $\forall$ $n \geq n_T$

Vậy với $\forall$ $n \geq \max(n_T, n_k)$ ta có $k(n).T(n) \leq c_T.c_k(g(n).f(n))$

2.2.3. Một số tính chất

Theo định nghĩa về độ phức tạp tính toán ta có một số tính chất:

a) Với P(n) là một đa thức bậc k thì $O(P(n)) = O(n^k)$. Vì thế, một thuật toán có độ phức tạp cấp đa thức, người ta thường ký hiệu là $O(n^k)$

b) Với a và b là hai cơ số tuỳ ý và f(n) là một hàm dương thì $\log_a f(n) = \log_a b.\log_b f(n)$. Tức là: $O(\log_a f(n)) = O(\log_b f(n))$. Vậy với một thuật toán có độ phức tạp cấp logarit của f(n), người ta ký hiệu là $O(\log f(n))$ mà không cần ghi cơ số của logarit.

c) Nếu một thuật toán có độ phức tạp là hằng số, tức là thời gian thực hiện không phụ thuộc vào kích thước dữ liệu vào thì ta ký hiệu độ phức tạp tính toán của thuật toán đó là O(1).

d) Một giải thuật có cấp là các hàm như 2^n, n!, n^n được gọi là một giải thuật có độ phức tạp hàm mũ. Những giải thuật như vậy trên thực tế thường có tốc độ rất chậm. Các giải thuật có cấp là các hàm đa thức hoặc nhỏ hơn hàm đa thức thì thường chấp nhận được.

e) Không phải lúc nào một giải thuật cấp $O(n^2)$ cũng tốt hơn giải thuật cấp $O(n^3)$. Bởi nếu như giải thuật cấp $O(n^2)$ có thời gian thực hiện là $1000n^2$, còn giải thuật cấp $O(n^3)$ lại chỉ cần thời

gian thực hiện là n^3, thì với n < 1000, rõ ràng giải thuật $O(n^3)$ tốt hơn giải thuật $O(n^2)$. Trên đây là xét trên phương diện tính toán lý thuyết để định nghĩa giải thuật này "tốt" hơn giải thuật kia, khi chọn một thuật toán để giải một bài toán thực tế phải có một sự mềm dẻo nhất định.

f) Cũng theo định nghĩa về độ phức tạp tính toán

Một thuật toán có cấp O(1) cũng có thể viết là O(logn)

Một thuật toán có cấp O(logn) cũng có thể viết là O(n)

Một thuật toán có cấp O(n) cũng có thể viết là O(n.logn)

Một thuật toán có cấp O(n.logn) cũng có thể viết là $O(n^2)$

Một thuật toán có cấp $O(n^2)$ cũng có thể viết là $O(n^3)$

Một thuật toán có cấp $O(n^3)$ cũng có thể viết là $O(2^n)$

Vậy độ phức tạp tính toán của một thuật toán có nhiều cách ký hiệu, thông thường người ta chọn cấp thấp nhất có thể, tức là chọn ký pháp O(f(n)) với f(n) là một hàm tăng chậm nhất theo n.

Dưới đây là một số hàm số hay dùng để ký hiệu độ phức tạp tính toán và bảng giá trị của chúng để tiện theo dõi sự tăng của hàm theo đối số n.

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2147483648

Ví dụ:

Thuật toán tính tổng các số từ 1 tới n:

Nếu viết theo sơ đồ như sau:

```
Input n;
S := 0;
for i := 1 to n do S := S + i;
Output S;
```

Các đoạn chương trình ở các dòng 1, 2 và 4 có độ phức tạp tính toán là O(1).

Vòng lặp ở dòng 3 lặp n lần phép gán S := S + i, nên thời gian tính toán tỉ lệ thuận với n. Tức là độ phức tạp tính toán là O(n).

Vậy độ phức tạp tính toán của thuật toán trên là O(n).

Còn nếu viết theo sơ đồ như sau:

```
Input n;
S := n * (n - 1) div 2;
Output S;
```

Thì độ phức tạp tính toán của thuật toán trên là O(1), thời gian tính toán không phụ thuộc vào n.

2.2.4. Phép toán tích cực

Dựa vào những nhận xét đã nêu ở trên về các quy tắc khi đánh giá thời gian thực hiện giải thuật, ta chỉ cần chú ý đến một phép toán mà ta gọi là phép toán tích cực trong một đoạn chương trình. Đó là **một phép toán trong một đoạn chương trình mà số lần thực hiện không ít hơn các phép toán khác**.

Xét hai đoạn chương trình tính e^x bằng công thức gần đúng:

$$e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!} = \sum_{i=0}^{n} \frac{x^i}{i!}$$ với x và n cho trước.

```
{Chương trình 1: Tính riêng từng hạng tử rồi cộng lại}
program Exp1;
var
 i, j, n: Integer;
 x, p, S: Real;
begin
 Write('x, n = '); ReadLn(x, n);
 S := 0;
 for i := 0 to n do
  begin
   p := 1;
   for j := 1 to i do p := p * x / j;
   S := S + p;
  end;
 WriteLn('exp(', x:1:4, ') = ', S:1:4);
end.
```

Ta có thể coi phép toán tích cực ở đây là

p := p * x / j;

Số lần thực hiện phép toán này là:

0 + 1 + 2 + … + n = n(n - 1)/2 lần.

Vậy độ phức tạp tính toán của thuật toán là $O(n^2)$

```
{Tính hạng tử sau qua hạng tử trước}
program Exp2;
var
 i, n: Integer;
 x, p, S: Real;
begin
 Write('x, n = '); ReadLn(x, n);
 S := 1; p := 1;
 for i := 1 to n do
  begin
   p := p * x / i;
   S := S + p;
  end;
 WriteLn('exp(', x:1:4, ') = ', S:1:4);
end.
```

Ta có thể coi phép toán tích cực ở đây là phép

p := p * x / i.

Số lần thực hiện phép toán này là n.

Vậy độ phức tạp tính toán của thuật toán là O(n).

2.3. ĐỘ PHỨC TẠP TÍNH TOÁN VỚI TÌNH TRẠNG DỮ LIỆU VÀO

Có nhiều trường hợp, thời gian thực hiện giải thuật không phải chỉ phụ thuộc vào kích thước dữ liệu mà còn phụ thuộc vào tình trạng của dữ liệu đó nữa. Chẳng hạn thời gian sắp xếp một dãy số theo thứ tự tăng dần mà dãy đưa vào chưa có thứ tự sẽ khác với thời gian sắp xếp một dãy số đã sắp xếp rồi hoặc đã sắp xếp theo thứ tự ngược lại. Lúc này, khi phân tích thời gian thực hiện giải thuật ta sẽ phải xét tới trường hợp tốt nhất, trường hợp trung bình và trường hợp xấu nhất. Khi khó khăn trong việc xác định độ phức tạp tính toán trong trường hợp trung bình (bởi việc xác định T(n) trung bình thường phải dùng tới những công cụ toán phức tạp), người ta thường chỉ đánh giá độ phức tạp tính toán trong trường hợp xấu nhất.

2.4. CHI PHÍ THỰC HIỆN THUẬT TOÁN

Khái niệm độ phức tạp tính toán đặt ra là để đánh giá chi phí thực hiện một giải thuật về mặt thời gian. Nhưng chi phí thực hiện giải thuật còn có rất nhiều yếu tố khác nữa: không gian bộ nhớ phải sử dụng là một ví dụ. Tuy nhiên, trên phương diện phân tích lý thuyết, ta chỉ có thể

xét tới vấn đề thời gian bởi việc xác định các chi phí khác nhiều khi rất mơ hồ và phức tạp. Đối với người lập trình thì khác, một thuật toán với độ phức tạp dù rất thấp cũng sẽ là vô dụng nếu như không thể cài đặt được trên máy tính, chính vì vậy khi bắt tay cài đặt một thuật toán, ta phải biết cách tổ chức dữ liệu một cách khoa học, tránh lãng phí bộ nhớ không cần thiết. Có một quy luật tương đối khi tổ chức dữ liệu: Tiết kiệm được bộ nhớ thì thời gian thực hiện thường sẽ chậm hơn và ngược lại. Biết cân đối, dung hoà hai yếu tố đó là một kỹ năng cần thiết của người lập trình.

Bài tập

Bài 1

Chứng minh một cách chặt chẽ: Tại sao với P(n) là đa thức bậc k thì một giải thuật cấp O(P(n)) cũng có thể coi là cấp $O(n^k)$

Bài 2

Xác định độ phức tạp tính toán của những giải thuật sau bằng ký pháp chữ O lớn:

a) Đoạn chương trình tính tổng hai đa thức:

$P(x) = a_mx^m + a_{m-1}x^{m-1} + \ldots + a_1x + a_0$ và $Q(x) = b_nx^n + a_{n-1}x^{n-1} + \ldots + b_1x + b_0$

Để được đa thức

$R(x) = cpxp + cp_{-1}xp^{-1} + \ldots + c_1x + c_0$

```
if m < n then p := m else p := n; {p = min(m, n)}
for i := 0 to p do c[i] := a[i] + b[i];
if p < m then
  for i := p + 1 to m do c[i] := a[i]
else
  for i := p + 1 to n do c[i] := b[i];
while (p > 0) and (c[p] = 0) do p := p - 1;
```

b) Đoạn chương trình tính tích hai đa thức:

$P(x) = a_mx^m + a_{m-1}x^{m-1} + \ldots + a_1x + a_0$ và $Q(x) = b_nx^n + a_{n-1}x^{n-1} + \ldots + b_1x + b_0$

Để được đa thức

$R(x) = c_px^p + c_{p-1}x^{p-1} + \ldots + c_1x + c_0$

```
p := m + n;
for i := 0 to p do c[i] := 0;
for i := 0 to m do
  for j := 0 to n do
    c[i + j] := c[i + j] + a[i] * b[j];
```

§3. ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

3.1. KHÁI NIỆM VỀ ĐỆ QUY

Ta nói một đối tượng là đệ quy nếu nó được định nghĩa qua chính nó hoặc một đối tượng khác cùng dạng với chính nó bằng quy nạp.

Ví dụ: Đặt hai chiếc gương cầu đối diện nhau. Trong chiếc gương thứ nhất chứa hình chiếc gương thứ hai. Chiếc gương thứ hai lại chứa hình chiếc gương thứ nhất nên tất nhiên nó chứa lại hình ảnh của chính nó trong chiếc gương thứ nhất… Ở một góc nhìn hợp lý, ta có thể thấy một dãy ảnh vô hạn của cả hai chiếc gương.

Một ví dụ khác là nếu người ta phát hình trực tiếp phát thanh viên ngồi bên máy vô tuyến truyền hình, trên màn hình của máy này lại có chính hình ảnh của phát thanh viên đó ngồi bên máy vô tuyến truyền hình và cứ như thế…

Trong toán học, ta cũng hay gặp các định nghĩa đệ quy:

Giai thừa của n (n!): Nếu n = 0 thì n! = 1; nếu n > 0 thì n! = n.(n-1)!

Ký hiệu số phần tử của một tập hợp hữu hạn S là |S|: Nếu S = ∅ thì |S| = 0; Nếu S ≠ ∅ thì tất có một phần tử x ∈ S, khi đó |S| = |S\{x}| + 1. Đây là phương pháp định nghĩa tập các số tự nhiên.

3.2. GIẢI THUẬT ĐỆ QUY

Nếu lời giải của một bài toán P được thực hiện bằng lời giải của bài toán P' có dạng giống như P thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy. Mới nghe thì có vẻ hơi lạ nhưng điểm mấu chốt cần lưu ý là: P' tuy có dạng giống như P, nhưng theo một nghĩa nào đó, nó phải "nhỏ" hơn P, dễ giải hơn P và việc giải nó không cần dùng đến P.

Trong Pascal, ta đã thấy nhiều ví dụ của các hàm và thủ tục có chứa lời gọi đệ quy tới chính nó, bây giờ, ta tóm tắt lại các phép đệ quy trực tiếp và tương hỗ được viết như thế nào:

Định nghĩa một hàm đệ quy hay thủ tục đệ quy gồm hai phần:

Phần neo (anchor): Phần này được thực hiện khi mà công việc quá đơn giản, có thể giải trực tiếp chứ không cần phải nhờ đến một bài toán con nào cả.

Phần đệ quy: Trong trường hợp bài toán chưa thể giải được bằng phần neo, ta xác định những bài toán con và gọi đệ quy giải những bài toán con đó. Khi đã có lời giải (đáp số) của những bài toán con rồi thì phối hợp chúng lại để giải bài toán đang quan tâm.

Phần đệ quy thể hiện tính "quy nạp" của lời giải. Phần neo cũng rất quan trọng bởi nó quyết định tới tính hữu hạn dừng của lời giải.

3.3. VÍ DỤ VỀ GIẢI THUẬT ĐỆ QUY

3.3.1. Hàm tính giai thừa

```
function Factorial(n: Integer): Integer; {Nhận vào số tự nhiên n và trả về n!}
begin
  if n = 0 then Factorial := 1 {Phần neo}
  else Factorial := n * Factorial(n - 1); {Phần đệ quy}
end;
```

Ở đây, phần neo định nghĩa kết quả hàm tại n = 0, còn phần đệ quy (ứng với n > 0) sẽ định nghĩa kết quả hàm qua giá trị của n và giai thừa của n - 1.

Ví dụ: Dùng hàm này để tính 3!, trước hết nó phải đi tính 2! bởi 3! được tính bằng tích của 3 * 2!. Tương tự để tính 2!, nó lại đi tính 1! bởi 2! được tính bằng 2 * 1!. Áp dụng bước quy nạp này thêm một lần nữa, 1! = 1 * 0!, và ta đạt tới trường hợp của phần neo, đến đây từ giá trị 1 của 0!, nó tính được 1! = 1*1 = 1; từ giá trị của 1! nó tính được 2!; từ giá trị của 2! nó tính được 3!; cuối cùng cho kết quả là 6:

```
3! = 3 * 2!
        ↓
        2! = 2 * 1!
                ↓
                1! = 1 * 0!
                        ↓
                        0! = 1
```

3.3.2. Dãy số Fibonacci

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán đặt ra như sau:

1) Các con thỏ không bao giờ chết

2) Hai tháng sau khi ra đời, mỗi cặp thỏ mới sẽ sinh ra một cặp thỏ con (một đực, một cái)

3) Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp con mới

Giả sử từ đầu tháng 1 có một cặp mới ra đời thì đến giữa tháng thứ n sẽ có bao nhiêu cặp.

Ví dụ, n = 5, ta thấy:

Giữa tháng thứ 1:1 cặp (ab) (cặp ban đầu)

Giữa tháng thứ 2:1 cặp (ab) (cặp ban đầu vẫn chưa đẻ)

Giữa tháng thứ 3:2 cặp (AB)(cd) (cặp ban đầu đẻ ra thêm 1 cặp con)

Giữa tháng thứ 4:3 cặp (AB)(cd)(ef) (cặp ban đầu tiếp tục đẻ)

Giữa tháng thứ 5:5 cặp (AB)(CD)(ef)(gh)(ik) (cả cặp (AB) và (CD) cùng đẻ)

Bây giờ, ta xét tới việc tính số cặp thỏ ở tháng thứ n: F(n)

Nếu mỗi cặp thỏ ở tháng thứ n - 1 đều sinh ra một cặp thỏ con thì số cặp thỏ ở tháng thứ n sẽ là:

$$F(n) = 2 * F(n - 1)$$

Nhưng vấn đề không phải như vậy, trong các cặp thỏ ở tháng thứ n - 1, chỉ có những cặp thỏ đã có ở tháng thứ n - 2 mới sinh con ở tháng thứ n được thôi. Do đó F(n) = F(n - 1) + F(n - 2) (= số cũ + số sinh ra). Vậy có thể tính được F(n) theo công thức sau:

F(n) = 1 nếu n ≤ 2

F(n) = F(n - 1) + F(n - 2) nếu n > 2

```
function F(n: Integer): Integer; {Tính số cặp thỏ ở tháng thứ n}
begin
  if n ≤ 2 then F := 1 {Phần neo}
  else F := F(n - 1) + F(n - 2); {Phần đệ quy}
end;
```

3.3.3. Giả thuyết của Collatz

Collatz đưa ra giả thuyết rằng: với một số nguyên dương X, nếu X chẵn thì ta gán X := X div 2; nếu X lẻ thì ta gán X := X * 3 + 1. Thì sau một số hữu hạn bước, ta sẽ có X = 1.

Ví du: X = 10, các bước tiến hành như sau:

```
1.  X = 10 (chẵn)  ⇒  X := 10 div 2;    (5)
2.  X = 5 (lẻ)     ⇒  X := 5 * 3 + 1;  (16)
3.  X = 16 (chẵn)  ⇒  X := 16 div 2;    (8)
4.  X = 8 (chẵn)   ⇒  X := 8 div 2      (4)
5.  X = 4 (chẵn)   ⇒  X := 4 div 2      (2)
6.  X = 2 (chẵn)   ⇒  X := 2 div 2      (1)
```

Cứ cho giả thuyết Collatz là đúng đắn, vấn đề đặt ra là: Cho trước số 1 cùng với hai phép toán * 2 và div 3, hãy sử dụng một cách hợp lý hai phép toán đó để biến số 1 thành một giá trị nguyên dương X cho trước.

Ví dụ: X = 10 ta có 1 * 2 * 2 * 2 * 2 div 3 * 2 = 10.

Dễ thấy rằng lời giải của bài toán gần như thứ tự ngược của phép biến đổi Collatz: Để biểu diễn số X > 1 bằng một biểu thức bắt đầu bằng số 1 và hai phép toán "* 2", "div 3". Ta chia hai trường hợp:

Nếu X chẵn, thì ta tìm cách biểu diễn số X div 2 và viết thêm phép toán * 2 vào cuối

Nếu X lẻ, thì ta tìm cách biểu diễn số X * 3 + 1 và viết thêm phép toán div 3 vào cuối

```
procedure Solve(X: Integer); {In ra cách biểu diễn số X}
begin
  if X = 1 then Write(X) {Phần neo}
  else {Phần đệ quy}
    if X mod 2 = 0 then {X chẵn}
      begin
       Solve(X div 2); {Tìm cách biểu diễn số X div 2}
       Write(' * 2'); {Sau đó viết thêm phép toán * 2}
      end
    else {X lẻ}
      begin
        Solve(X * 3 + 1); {Tìm cách biểu diễn số X * 3 + 1}
        Write(' div 3'); {Sau đó viết thêm phép toán div 3}
      end;
end;
```

Trên đây là cách viết đệ quy trực tiếp, còn có một cách viết đệ quy tương hỗ như sau:

```
procedure Solve(X: Integer); forward; {Thủ tục tìm cách biểu diễn số X: Khai báo trước, đặc tả sau}
```

```
procedure SolveOdd(X: Integer); {Thủ tục tìm cách biểu diễn số X > 1 trong trường hợp X lẻ}
begin
  Solve(X * 3 + 1);
  Write(' div 3');
end;

procedure SolveEven(X: Integer); {Thủ tục tìm cách biểu diễn số X trong trường hợp X chẵn}
begin
  Solve(X div 2);
  Write(' * 2');
end;

procedure Solve(X: Integer); {Phần đặc tả của thủ tục Solve đã khai báo trước ở trên}
begin
  if X = 1 then Write(X)
  else
    if X mod 2 = 1 then SolveOdd(X)
    else SolveEven(X);
end;
```

Trong cả hai cách viết, để tìm biểu diễn số X theo yêu cầu chỉ cần gọi Solve(X) là xong. Tuy nhiên trong cách viết đệ quy trực tiếp, thủ tục Solve có lời gọi tới chính nó, còn trong cách viết đệ quy tương hỗ, thủ tục Solve chứa lời gọi tới thủ tục SolveOdd và SolveEven, hai thủ tục này lại chứa trong nó lời gọi ngược về thủ tục Solve.

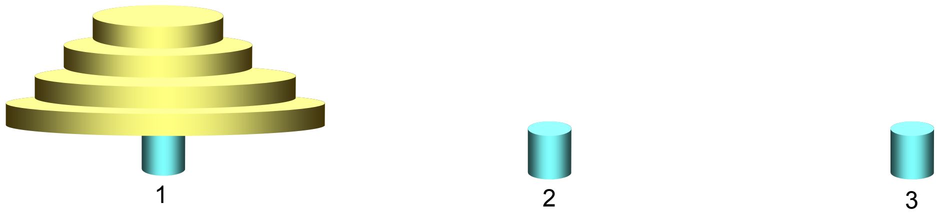
Đối với những bài toán nêu trên, việc thiết kế các giải thuật đệ quy tương ứng khá thuận lợi vì cả hai đều thuộc dạng tính giá trị hàm mà định nghĩa quy nạp của hàm đó được xác định dễ dàng.

Nhưng không phải lúc nào phép giải đệ quy cũng có thể nhìn nhận và thiết kế dễ dàng như vậy. Thế thì vấn đề gì cần lưu tâm trong phép giải đệ quy?. Có thể tìm thấy câu trả lời qua việc giải đáp các câu hỏi sau:

1. Có thể định nghĩa được bài toán dưới dạng phối hợp của những bài toán cùng loại nhưng nhỏ hơn hay không ? Khái niệm "nhỏ hơn" là thế nào ?
2. Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp tầm thường và có thể giải ngay được để đưa vào phần neo của phép giải đệ quy

3.3.4. Bài toán Tháp Hà Nội

Đây là một bài toán mang tính chất một trò chơi, tương truyền rằng tại ngôi đền Benares có ba cái cọc kim cương. Khi khai sinh ra thế giới, thượng đế đặt n cái đĩa bằng vàng chồng lên nhau theo thứ tự giảm dần của đường kính tính từ dưới lên, đĩa to nhất được đặt trên một chiếc cọc.



Hình 5: Tháp Hà Nội

Các nhà sư lần lượt chuyển các đĩa sang cọc khác theo luật:

- Khi di chuyển một đĩa, phải đặt nó vào một trong ba cọc đã cho
- Mỗi lần chỉ có thể chuyển một đĩa và phải là đĩa ở trên cùng
- Tại một vị trí, đĩa nào mới chuyển đến sẽ phải đặt lên trên cùng
- Đĩa lớn hơn không bao giờ được phép đặt lên trên đĩa nhỏ hơn (hay nói cách khác: một đĩa chỉ được đặt trên cọc hoặc đặt trên một đĩa lớn hơn)

Ngày tận thế sẽ đến khi toàn bộ chồng đĩa được chuyển sang một cọc khác.

Trong trường hợp có 2 đĩa, cách làm có thể mô tả như sau:

Chuyển đĩa nhỏ sang cọc 3, đĩa lớn sang cọc 2 rồi chuyển đĩa nhỏ từ cọc 3 sang cọc 2.

Những người mới bắt đầu có thể giải quyết bài toán một cách dễ dàng khi số đĩa là ít, nhưng họ sẽ gặp rất nhiều khó khăn khi số các đĩa nhiều hơn. Tuy nhiên, với tư duy quy nạp toán học và một máy tính thì công việc trở nên khá dễ dàng:

Có n đĩa.

- Nếu n = 1 thì ta chuyển đĩa duy nhất đó từ cọc 1 sang cọc 2 là xong.
- Giả sử rằng ta có phương pháp chuyển được n - 1 đĩa từ cọc 1 sang cọc 2, thì cách chuyển n - 1 đĩa từ cọc x sang cọc y ($1 \le x, y \le 3$) cũng tương tự.
- Giả sử ràng ta có phương pháp chuyển được n - 1 đĩa giữa hai cọc bất kỳ. Để chuyển n đĩa từ cọc x sang cọc y, ta gọi cọc còn lại là z (=6 - x - y). Coi đĩa to nhất là … cọc, chuyển n - 1 đĩa còn lại từ cọc x sang cọc z, sau đó chuyển đĩa to nhất đó sang cọc y và cuối cùng lại coi đĩa to nhất đó là cọc, chuyển n - 1 đĩa còn lại đang ở cọc z sang cọc y chồng lên đĩa to nhất.

Cách làm đó được thể hiện trong thủ tục đệ quy dưới đây:

```
procedure Move(n, x, y: Integer); {Thủ tục chuyển n đĩa từ cọc x sang cọc y}
begin
  if n = 1 then WriteLn('Chuyển 1 đĩa từ ', x, ' sang ', y)
  else {Để chuyển n > 1 đĩa từ cọc x sang cọc y, ta chia làm 3 công đoạn}
    begin
      Move(n - 1, x, 6 - x - y); {Chuyển n - 1 đĩa từ cọc x sang cọc trung gian}
      Move(1, x, y); {Chuyển đĩa to nhất từ x sang y}
      Move(n - 1, 6 - x - y, y); {Chuyển n - 1 đĩa từ cọc trung gian sang cọc y}
    end;
end;
```

Chương trình chính rất đơn giản, chỉ gồm có 2 việc: Nhập vào số n và gọi Move(n, 1, 2).

3.4. HIỆU LỰC CỦA ĐỆ QUY

Qua các ví dụ trên, ta có thể thấy đệ quy là một công cụ mạnh để giải các bài toán. Có những bài toán mà bên cạnh giải thuật đệ quy vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn bài toán tính giai thừa hay tính số Fibonacci. Tuy vậy, đệ quy vẫn có vai trò xứng đáng của nó, có nhiều bài toán mà việc thiết kế giải thuật đệ quy đơn giản hơn nhiều so với lời giải lặp và trong một số trường hợp chương trình đệ quy hoạt động nhanh hơn chương trình viết không có đệ quy. Giải thuật cho bài Tháp Hà Nội và thuật toán sắp xếp kiểu phân đoạn (QuickSort) mà ta sẽ nói tới trong các bài sau là những ví dụ.

Có một mối quan hệ khăng khít giữa đệ quy và quy nạp toán học. Cách giải đệ quy cho một bài toán dựa trên việc định rõ lời giải cho trường hợp suy biến (neo) rồi thiết kế làm sao để lời giải của bài toán được suy ra từ lời giải của bài toán nhỏ hơn cùng loại như thế. Tương tự như vậy, quy nạp toán học chứng minh một tính chất nào đó ứng với số tự nhiên cũng bằng cách chứng minh tính chất đó đúng với một số trường hợp cơ sở (thường người ta chứng minh nó đúng với 0 hay đúng với 1) và sau đó chứng minh tính chất đó sẽ đúng với n bất kỳ nếu nó đã đúng với mọi số tự nhiên nhỏ hơn n.

Do đó ta không lấy làm ngạc nhiên khi thấy quy nạp toán học được dùng để chứng minh các tính chất có liên quan tới giải thuật đệ quy. Chẳng hạn: Chứng minh số phép chuyển đĩa để giải bài toán Tháp Hà Nội với n đĩa là 2^n-1:

Rõ ràng là tính chất này đúng với n = 1, bởi ta cần $2^1 - 1 = 1$ lần chuyển đĩa để thực hiện yêu cầu

Với n > 1; Giả sử rằng để chuyển n - 1 đĩa giữa hai cọc ta cần $2^{n-1} - 1$ phép chuyển đĩa, khi đó để chuyển n đĩa từ cọc x sang cọc y, nhìn vào giải thuật đệ quy ta có thể thấy rằng trong trường hợp này nó cần $(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$ phép chuyển đĩa. Tính chất được chứng minh đúng với n

Vậy thì công thức này sẽ đúng với mọi n.

Thật đáng tiếc nếu như chúng ta phải lập trình với một công cụ không cho phép đệ quy, nhưng như vậy không có nghĩa là ta bó tay trước một bài toán mang tính đệ quy. Mọi giải thuật đệ quy đều có cách thay thế bằng một giải thuật không đệ quy (khử đệ quy), có thể nói được như vậy bởi tất cả các chương trình con đệ quy sẽ đều được trình dịch chuyển thành những mã lệnh không đệ quy trước khi giao cho máy tính thực hiện.

Việc tìm hiểu cách khử đệ quy một cách "máy móc" như các chương trình dịch thì chỉ cần hiểu rõ cơ chế xếp chồng của các thủ tục trong một dây chuyền gọi đệ quy là có thể làm được. Nhưng muốn khử đệ quy một cách tinh tế thì phải tuỳ thuộc vào từng bài toán mà khử đệ quy cho khéo. Không phải tìm đâu xa, những kỹ thuật giải công thức truy hồi bằng quy hoạch

động là ví dụ cho thấy tính nghệ thuật trong những cách tiếp cận bài toán mang bản chất đệ quy để tìm ra một giải thuật không đệ quy đầy hiệu quả.

Bài tập

Bài 1

Viết một hàm đệ quy tính ước số chung lớn nhất của hai số tự nhiên a, b không đồng thời bằng 0, chỉ rõ đâu là phần neo, đâu là phần đệ quy.

Bài 2

Viết một hàm đệ quy tính C_n^k theo công thức truy hồi sau:

$C_n^0 = C_n^n = 1$

Với 0 < k < n: $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$

Chứng minh rằng hàm đó cho ra đúng giá trị $C_n^k = \frac{n!}{k!(n-k)!}$.

Bài 3

Nêu rõ các bước thực hiện của giải thuật cho bài Tháp Hà Nội trong trường hợp n = 3.

Viết chương trình giải bài toán Tháp Hà Nội không đệ quy

§4. CẤU TRÚC DỮ LIỆU BIỂU DIỄN DANH SÁCH

4.1. KHÁI NIỆM DANH SÁCH

Danh sách là một tập sắp thứ tự các phần tử cùng một kiểu. Đối với danh sách, người ta có một số thao tác: Tìm một phần tử trong danh sách, chèn một phần tử vào danh sách, xoá một phần tử khỏi danh sách, sắp xếp lại các phần tử trong danh sách theo một trật tự nào đó v.v…

4.2. BIỂU DIỄN DANH SÁCH TRONG MÁY TÍNH

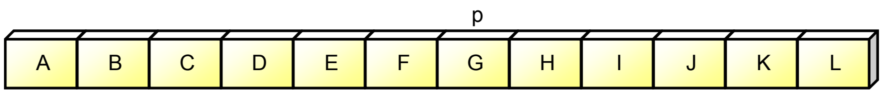
Việc cài đặt một danh sách trong máy tính tức là tìm một cấu trúc dữ liệu cụ thể mà máy tính hiểu được để lưu các phần tử của danh sách đồng thời viết các đoạn chương trình con mô tả các thao tác cần thiết đối với danh sách.

4.2.1. Cài đặt bằng mảng một chiều

Khi cài đặt danh sách bằng một mảng, thì có một biến nguyên n lưu số phần tử hiện có trong danh sách. Nếu mảng được đánh số bắt đầu từ 1 thì các phần tử trong danh sách được cất giữ trong mảng bằng các phần tử được đánh số từ 1 tới n.

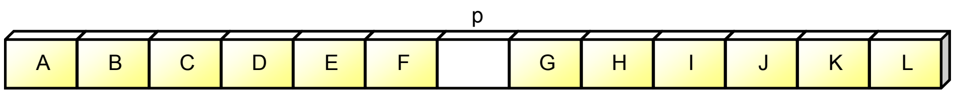
Chèn phần tử vào mảng:

Mảng ban đầu:

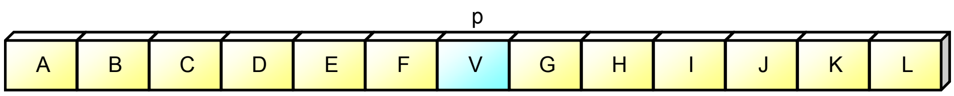


Nếu muốn chèn một phần tử V vào mảng tại vị trí p, ta phải:

Dồn tất cả các phần tử từ vị trí p tới tới vị trí n về sau một vị trí:



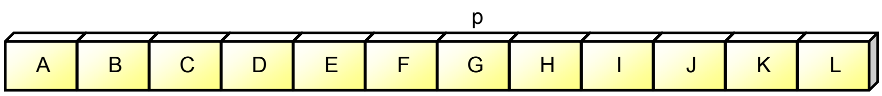
Đặt giá trị V vào vị trí p:



Tăng n lên 1

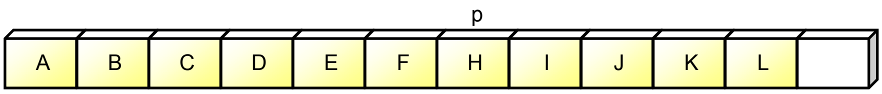
Xoá phần tử khỏi mảng

Mảng ban đầu:

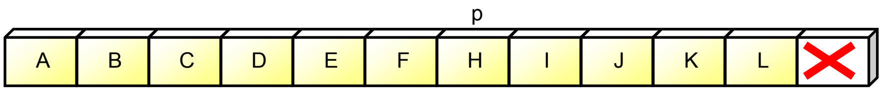


Muốn xoá phần tử thứ p của mảng mà vẫn giữ nguyên thứ tự các phần tử còn lại, ta phải:

Dồn tất cả các phần tử từ vị trí p + 1 tới vị trí n lên trước một vị trí:



Giảm n đi 1



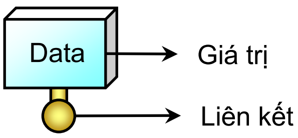
Trong trường hợp cần xóa một phần tử mà không cần duy trì thứ tự của các phần tử khác, ta chỉ cần đảo giá trị của phần tử cần xóa cho phần tử cuối cùng rồi giảm số phần tử của mảng (n) đi 1.

4.2.2. Cài đặt bằng danh sách nối đơn

Danh sách nối đơn gồm các nút được nối với nhau theo một chiều. Mỗi nút là một bản ghi (record) gồm hai trường:

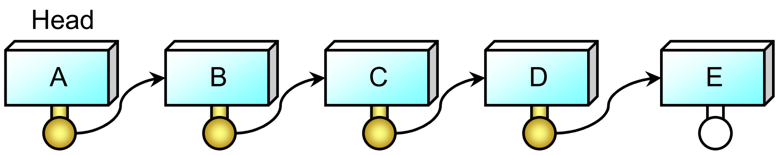
Trường thứ nhất chứa giá trị lưu trong nút đó

Trường thứ hai chứa liên kết (con trỏ) tới nút kế tiếp, tức là chứa một thông tin đủ để biết nút kế tiếp nút đó trong danh sách là nút nào, trong trường hợp là nút cuối cùng (không có nút kế tiếp), trường liên kết này được gán một giá trị đặc biệt.



Hình 6: Cấu trúc nút của danh sách nối đơn

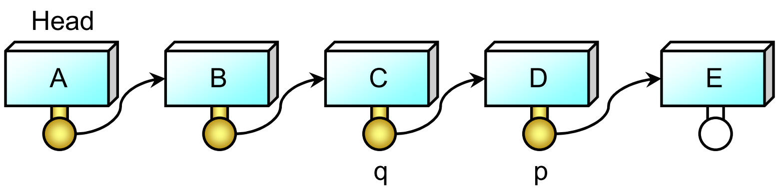
Nút đầu tiên trong danh sách được gọi là chốt của danh sách nối đơn (Head). Để duyệt danh sách nối đơn, ta bắt đầu từ chốt, dựa vào trường liên kết để đi sang nút kế tiếp, đến khi gặp giá trị đặc biệt (duyệt qua nút cuối) thì dừng lại



Hình 7: Danh sách nối đơn

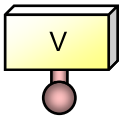
Chèn phần tử vào danh sách nối đơn:

Danh sách ban đầu:



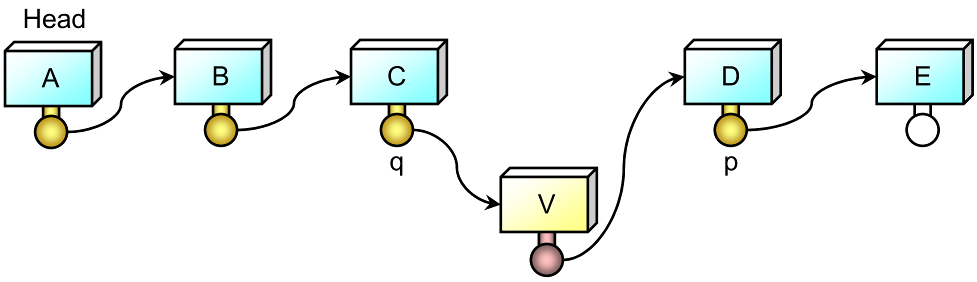
Muốn chèn thêm một nút chứa giá trị V vào vị trí của nút p, ta phải:

a) Tạo ra một nút mới NewNode chứa giá trị V:



b) Tìm nút q là nút đứng trước nút p trong danh sách (nút có liên kết tới p).

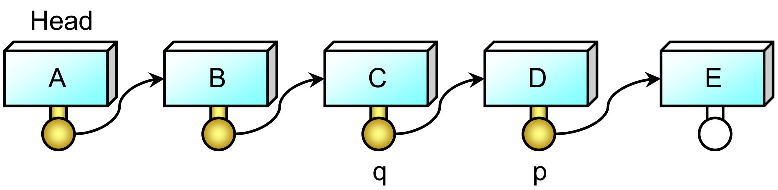
b_1) Nếu tìm thấy thì chỉnh lại liên kết: q liên kết tới NewNode, NewNode liên kết tới p



b_2) Nếu không có nút đứng trước nút p trong danh sách thì tức là p = Head, ta chỉnh lại liên kết: NewNode liên kết tới Head (cũ) và đặt lại Head = NewNode

Xoá phần tử khỏi danh sách nối đơn:

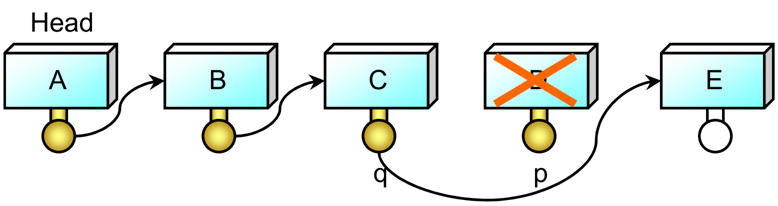
Danh sách ban đầu:



Muốn huỷ nút p khỏi danh sách nối đơn, ta phải:

Tìm nút q là nút đứng liền trước nút p trong danh sách (nút có liên kết tới p)

Nếu tìm thấy thì chỉnh lại liên kết: q liên kết thẳng tới nút liền sau p, khi đó quá trình duyệt danh sách bắt đầu từ Head khi duyệt tới q sẽ nhảy qua không duyệt p nữa. Trên thực tế khi cài đặt bằng các biến động và con trỏ, ta nên có thao tác giải phóng bộ nhớ đã cấp cho nút p

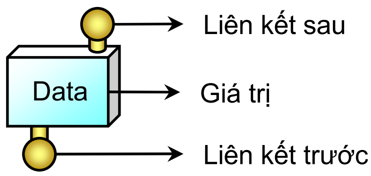


Nếu không có nút đứng trước nút p trong danh sách thì tức là p = Head, ta chỉ việc đặt lại Head bằng nút đứng kế tiếp Head (cũ) trong danh sách. Sau đó có thể giải phóng bộ nhớ cấp cho nút p (Head cũ)

4.2.3. Cài đặt bằng danh sách nối kép

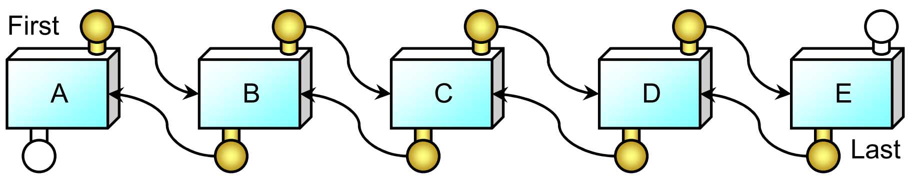
Danh sách nối kép gồm các nút được nối với nhau theo hai chiều. Mỗi nút là một bản ghi (record) gồm ba trường:

- Trường thứ nhất chứa giá trị lưu trong nút đó
- Trường thứ hai (Next) chứa liên kết (con trỏ) tới nút kế tiếp, tức là chứa một thông tin đủ để biết nút kế tiếp nút đó là nút nào, trong trường hợp là nút cuối cùng (không có nút kế tiếp), trường liên kết này được gán một giá tị đặc biệt.
- Trường thứ ba (Prev) chứa liên kết (con trỏ) tới nút liền trước, tức là chứa một thông tin đủ để biết nút đứng trước nút đó trong danh sách là nút nào, trong trường hợp là nút đầu tiên (không có nút liền trước) trường này được gán một giá trị đặc biệt.



Hình 8: Cấu trúc nút của danh sách nối kép

Khác với danh sách nối đơn, danh sách nối kép có hai chốt: Nút đầu tiên trong danh sách được gọi là First, nút cuối cùng trong danh sách được gọi là Last. Để duyệt danh sách nối kép, ta có hai cách: Hoặc bắt đầu từ First, dựa vào liên kết Next để đi sang nút kế tiếp, đến khi gặp giá trị đặc biệt (duyệt qua nút cuối) thì dừng lại. Hoặc bắt đầu từ Last, dựa vào liên kết Prev để đi sang nút liền trước, đến khi gặp giá trị đặc biệt (duyệt qua nút đầu) thì dừng lại

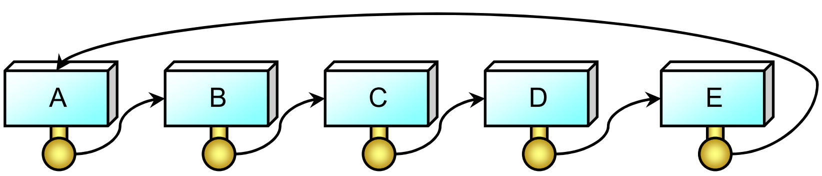


Hình 9: Danh sách nối kép

Việc chèn / xoá vào danh sách nối kép cũng đơn giản chỉ là kỹ thuật chỉnh lại các mối liên kết giữa các nút cho hợp lý, ta coi như bài tập.

4.2.4. Cài đặt bằng danh sách nối vòng một hướng

Trong danh sách nối đơn, phần tử cuối cùng trong danh sách có trường liên kết được gán một giá trị đặc biệt (thường sử dụng nhất là giá trị nil). Nếu ta cho trường liên kết của phần tử cuối cùng trỏ thẳng về phần tử đầu tiên của danh sách thì ta sẽ được một kiểu danh sách mới gọi là danh sách nối vòng một hướng.

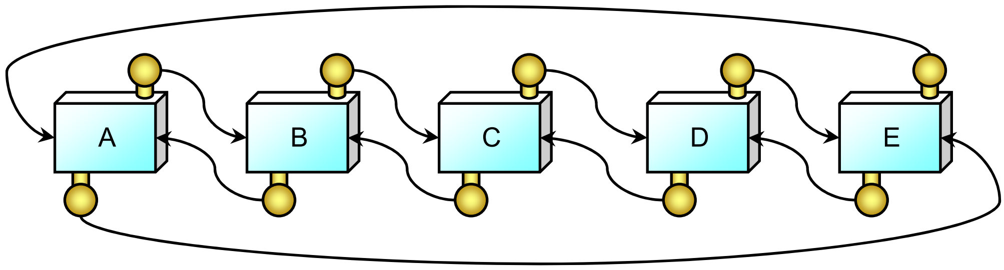


Hình 10: Danh sách nối vòng một hướng

Đối với danh sách nối vòng, ta chỉ cần biết một nút bất kỳ của danh sách là ta có thể duyệt được hết các nút trong danh sách bằng cách đi theo hướng của các liên kết. Chính vì lý do này, khi chèn xoá vào danh sách nối vòng, ta không phải xử lý các trường hợp riêng khi chèn xoá tại vị trí của chốt

4.2.5. Cài đặt bằng danh sách nối vòng hai hướng

Danh sách nối vòng một hướng chỉ cho ta duyệt các nút của danh sách theo một chiều, nếu cài đặt bằng danh sách nối vòng hai hướng thì ta có thể duyệt các nút của danh sách cả theo chiều ngược lại nữa. Danh sách nối vòng hai hướng có thể tạo thành từ danh sách nối kép nếu ta cho trường Prev của nút First trỏ thẳng tới nút Last còn trường Next của nút Last thì trỏ thẳng về nút First.



Hình 11: Danh sách nối vòng hai hướng

Bài tập

Bài 1

Lập chương trình quản lý danh sách học sinh, tuỳ chọn loại danh sách cho phù hợp, chương trình có những chức năng sau: (Hồ sơ một học sinh giả sử có: Tên, lớp, số điện thoại, điểm TB …)

Cho phép nhập danh sách học sinh từ bàn phím hay từ file.

Cho phép in ra danh sách học sinh gồm có tên và xếp loại

Cho phép in ra danh sách học sinh gồm các thông tin đầy đủ

Cho phép nhập vào từ bàn phím một tên học sinh và một tên lớp, tìm xem có học sinh có tên nhập vào trong lớp đó không ?. Nếu có thì in ra số điện thoại của học sinh đó

Cho phép vào một hồ sơ học sinh mới từ bàn phím, bổ sung học sinh đó vào danh sách học sinh, in ra danh sách mới.

Cho phép nhập vào từ bàn phím tên một lớp, loại bỏ tất cả các học sinh của lớp đó khỏi danh sách, in ra danh sách mới.

Có chức năng sắp xếp danh sách học sinh theo thứ tự giảm dần của điểm trung bình

Cho phép nhập vào hồ sơ một học sinh mới từ bàn phím, chèn học sinh đó vào danh sách mà không làm thay đổi thứ tự đã sắp xếp, in ra danh sách mới.

Cho phép lưu trữ lại trên đĩa danh sách học sinh khi đã thay đổi.

Bài 2

Có n người đánh số từ 1 tới n ngồi quanh một vòng tròn ($n \leq 10000$), cùng chơi một trò chơi: Một người nào đó đếm 1, người kế tiếp, theo chiều kim đồng hồ đếm 2… cứ như vậy cho tới người đếm đến một số nguyên tố thì phải ra khỏi vòng tròn, người kế tiếp lại đếm bắt đầu từ 1: Hãy lập chương trình

Nhập vào 2 số n và S từ bàn phím

- Cho biết nếu người thứ nhất là người đếm 1 thì người còn lại cuối cùng trong vòng tròn là người thứ mấy
- Cho biết nếu người còn lại cuối cùng trong vòng tròn là người thứ k thì người đếm 1 là người nào?.

Giải quyết hai yêu cầu trên trong trường hợp: đầu tiên trò chơi được đếm theo chiều kim đồng hồ, khi có một người bị ra khỏi cuộc chơi thì vẫn là người kế tiếp đếm 1 nhưng quá trình đếm ngược lại (tức là ngược chiều kim đồng hồ)

§5. NGĂN XẾP VÀ HÀNG ĐỢI

5.1. NGĂN XẾP (STACK)

Ngăn xếp là một kiểu danh sách được trang bị hai phép toán **bổ sung một phần tử** vào cuối danh sách và **loại bỏ một phần tử** cũng ở cuối danh sách.

Có thể hình dung ngăn xếp như hình ảnh một chồng đĩa, đĩa nào được đặt vào chồng sau cùng sẽ nằm trên tất cả các đĩa khác và sẽ được lấy ra đầu tiên. Vì nguyên tắc"vào sau ra trước" đó, Stack còn có tên gọi là danh sách kiểu LIFO (Last In First Out) và vị trí cuối danh sách được gọi là đỉnh (Top) của Stack.

5.1.1. Mô tả Stack bằng mảng

Khi mô tả Stack bằng mảng:

Việc bổ sung một phần tử vào Stack tương đương với việc thêm một phần tử vào cuối mảng.

Việc loại bỏ một phần tử khỏi Stack tương đương với việc loại bỏ một phần tử ở cuối mảng.

Stack bị tràn khi bổ sung vào mảng đã đầy

Stack là rỗng khi số phần tử thực sự đang chứa trong mảng = 0.

```
program StackByArray;
const
  max = 10000;
var
  Stack: array[1..max] of Integer;
  Last: Integer;

procedure StackInit; {Khởi tạo Stack rỗng}
begin
  Last := 0;
end;

procedure Push(V: Integer); {Đẩy một giá trị V vào Stack}
begin
  if Last = max then WriteLn('Stack is full') {Nếu Stack đã đầy thì không đẩy được thêm vào nữa}
  else
    begin
      Inc(Last); Stack[Last] := V; {Nếu không thì thêm một phần tử vào cuối mảng}
    end;
end;

function Pop: Integer; {Lấy một giá trị ra khỏi Stack, trả về trong kết quả hàm}
begin
  if Last = 0 then WriteLn('Stack is empty') {Stack đang rỗng thì không lấy được}
  else
    begin
      Pop := Stack[Last]; Dec(Last); {Lấy phần tử cuối ra khỏi mảng}
    end;
end;

begin
  StackInit;
  <Test>; {Đưa một vài lệnh để kiểm tra hoạt động của Stack}
```

```
end.
```

Khi cài đặt bằng mảng, tuy các thao tác đối với Stack viết hết sức đơn giản nhưng ở đây ta vẫn chia thành các chương trình con, mỗi chương trình con mô tả một thao tác, để từ đó về sau, ta chỉ cần biết rằng chương trình của ta có một cấu trúc Stack, còn ta mô phỏng cụ thể như thế nào thì không cần phải quan tâm nữa, và khi cài đặt Stack bằng các cấu trúc dữ liệu khác, chỉ cần sửa lại các thủ tục StackInit, Push và Pop mà thôi.

5.1.2. Mô tả Stack bằng danh sách nối đơn kiểu LIFO

Khi cài đặt Stack bằng danh sách nối đơn kiểu LIFO, thì Stack bị tràn khi vùng không gian nhớ dùng cho các biến động không còn đủ để thêm một phần tử mới. Tuy nhiên, việc kiểm tra điều này rất khó bởi nó phụ thuộc vào máy tính và ngôn ngữ lập trình. Ví dụ như đối với Turbo Pascal, khi Heap còn trống 80 Bytes thì cũng chỉ đủ chỗ cho 10 biến, mỗi biến 6 Bytes mà thôi. Mặt khác, không gian bộ nhớ dùng cho các biến động thường rất lớn nên cài đặt dưới đây ta bỏ qua việc kiểm tra Stack tràn.

```
program StackByLinkedList;
type
  PNode = ^TNode; {Con trỏ tới một nút của danh sách}
  TNode = record {Cấu trúc một nút của danh sách}
    Value: Integer;
    Link: PNode;
  end;
var
  Last: PNode; {Con trỏ đỉnh Stack}

procedure StackInit; {Khởi tạo Stack rỗng}
begin
  Last := nil;
end;

procedure Push(V: Integer); {Đẩy giá trị V vào Stack ⇔ thêm nút mới chứa V và nối nút đó vào danh sách}
var
  P: PNode;
begin
  New(P); P^.Value := V; {Tạo ra một nút mới}
  P^.Link := Last; Last := P; {Móc nút đó vào danh sách}
end;

function Pop: Integer; {Lấy một giá trị ra khỏi Stack, trả về trong kết quả hàm}
var
  P: PNode;
begin
  if Last = nil then WriteLn('Stack is empty')
  else
    begin
      Pop := Last^.Value; {Gán kết quả hàm}
      P := Last^.Link; {Giữ lại nút tiếp theo last^ (nút được đẩy vào danh sách trước nút Last^)}
      Dispose(Last); Last := P; {Giải phóng bộ nhớ cấp cho Last^, cập nhật lại Last mới}
    end;
end;

begin
  StackInit;
  <Test>; {Đưa một vài lệnh để kiểm tra hoạt động của Stack}
```

```
end.
```

5.2. HÀNG ĐỢI (QUEUE)

Hàng đợi là một kiểu danh sách được trang bị hai phép toán **bổ sung một phần tử** vào cuối danh sách (Rear) và **loại bỏ một phần tử** ở đầu danh sách (Front).

Có thể hình dung hàng đợi như một đoàn người xếp hàng mua vé: Người nào xếp hàng trước sẽ được mua vé trước. Vì nguyên tắc"vào trước ra trước" đó, Queue còn có tên gọi là danh sách kiểu FIFO (First In First Out).

5.2.1. Mô tả Queue bằng mảng

Khi mô tả Queue bằng mảng, ta có hai chỉ số First và Last, First lưu chỉ số phần tử đầu Queue còn Last lưu chỉ số cuối Queue, khởi tạo Queue rỗng: First := 1 và Last := 0;

Để thêm một phần tử vào Queue, ta tăng Last lên 1 và đưa giá trị đó vào phần tử thứ Last.

Để loại một phần tử khỏi Queue, ta lấy giá trị ở vị trí First và tăng First lên 1.

Khi Last tăng lên hết khoảng chỉ số của mảng thì mảng đã đầy, không thể đẩy thêm phần tử vào nữa.

Khi First > Last thì tức là Queue đang rỗng

Như vậy chỉ một phần của mảng từ vị trí First tới Last được sử dụng làm Queue.

```
program QueueByArray;
const
  max = 10000;
var
  Queue: array[1..max] of Integer;
  First, Last: Integer;

procedure QueueInit; {Khởi tạo một hàng đợi rỗng}
begin
  First := 1; Last := 0;
end;

procedure Push(V: Integer); {Đẩy V vào hàng đợi}
begin
  if Last = max then WriteLn('Overflow')
  else
    begin
      Inc(Last);
      Queue[Last] := V;
    end;
end;

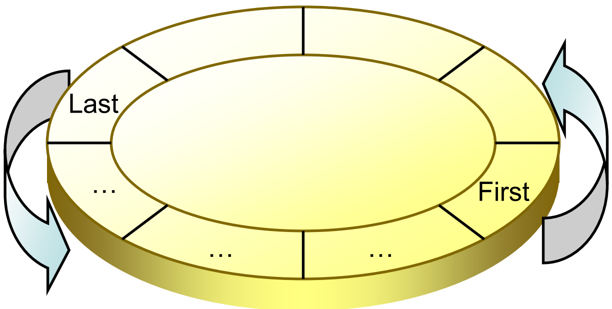
function Pop: Integer; {Lấy một giá trị khỏi hàng đợi, trả về trong kết quả hàm}
begin
  if First > Last then WriteLn('Queue is Empty')
  else
    begin
      Pop := Queue[First];
      Inc(First);
    end;
end;

begin
```

```
  QueueInit;
  <Test>; {Đưa một vài lệnh để kiểm tra hoạt động của Queue}
end.
```

Xem lại chương trình cài đặt Stack bằng một mảng kích thước tối đa 10000 phần tử, ta thấy rằng nếu như ta làm 6000 lần Push rồi 6000 lần Pop rồi lại 6000 lần Push thì vẫn không có vấn đề gì xảy ra. Lý do là vì chỉ số Last lưu đỉnh của Stack sẽ được tăng lên 6000 rồi lại giảm đến 0 rồi lại tăng trở lại lên 6000. Nhưng đối với cách cài đặt Queue như trên thì sẽ gặp thông báo lỗi tràn mảng, bởi mỗi lần Push, chỉ số cuối hàng đợi Last cũng tăng lên và không bao giờ bị giảm đi cả. Đó chính là nhược điểm mà ta nói tới khi cài đặt: Chỉ có các phần tử từ vị trí First tới Last là thuộc Queue, các phần tử từ vị trí 1 tới First - 1 là vô nghĩa.

Để khắc phục điều này, ta mô tả Queue bằng một danh sách vòng (biểu diễn bằng mảng hoặc cấu trúc liên kết), coi như các phần tử của mảng được xếp quanh vòng theo một hướng nào đó. Các phần tử nằm trên phần cung tròn từ vị trí First tới vị trí Last là các phần tử của Queue. Có thêm một biến n lưu số phần tử trong Queue. Việc thêm một phần tử vào Queue tương đương với việc ta dịch chỉ số Last theo vòng một vị trí rồi đặt giá trị mới vào đó. Việc loại bỏ một phần tử trong Queue tương đương với việc lấy ra phần tử tại vị trí First rồi dịch chỉ số First theo vòng.



Hình 12: Dùng danh sách vòng mô tả Queue

Lưu ý là trong thao tác Push và Pop phải kiểm tra Queue tràn hay Queue cạn nên phải cập nhật lại biến n. (Ở đây dùng thêm biến n cho dễ hiểu còn trên thực tế chỉ cần hai biến First và Last là ta có thể kiểm tra được Queue tràn hay cạn rồi)

```
program QueueByCList;
const
  max = 10000;
var
  Queue: array[0..max - 1] of Integer;
  i, n, First, Last: Integer;

procedure QueueInit; {Khởi tạo Queue rỗng}
begin
  First := 0; Last := max - 1; n := 0;
end;

procedure Push(V: Integer); {Đẩy giá trị V vào Queue}
begin
  if n = max then WriteLn('Queue is Full')
  else
```

```
    begin
      Last := (Last + 1) mod max; {Last chạy theo vòng tròn}
      Queue[Last] := V;
      Inc(n);
    end;
end;

function Pop: Integer; {Lấy một phần tử khỏi Queue, trả về trong kết quả hàm}
begin
  if n = 0 then WriteLn('Queue is Empty')
  else
    begin
      Pop := Queue[First];
      First := (First + 1) mod max; {First chạy theo vòng tròn}
      Dec(n);
    end;
end;

begin
  QueueInit;
  <Test>; {Đưa một vài lệnh để kiểm tra hoạt động của Queue}
end.
```

5.2.2. Mô tả Queue bằng danh sách nối đơn kiểu FIFO

Tương tự như cài đặt Stack bằng danh sách nối đơn kiểu LIFO, ta cũng không kiểm tra Queue tràn trong trường hợp mô tả Queue bằng danh sách nối đơn kiểu FIFO.

```
program QueueByLinkedList;
type
  PNode = ^TNode; {Kiểu con trỏ tới một nút của danh sách}
  TNode = record {Cấu trúc một nút của danh sách}
    Value: Integer;
    Link: PNode;
  end;
var
  First, Last: PNode; {Hai con trỏ tới nút đầu và nút cuối của danh sách}

procedure QueueInit; {Khởi tạo Queue rỗng}
begin
  First := nil;
end;

procedure Push(V: Integer); {Đẩy giá trị V vào Queue}
var
  P: PNode;
begin
  New(P); P^.Value := V; {Tạo ra một nút mới}
  P^.Link := nil;
  if First = nil then First := P {Móc nút đó vào danh sách}
  else Last^.Link := P;
  Last := P; {Nút mới trở thành nút cuối, cập nhật lại con trỏ Last}
end;

function Pop: Integer; {Lấy giá trị khỏi Queue, trả về trong kết quả hàm}
var
  P: PNode;
begin
  if First = nil then WriteLn('Queue is empty')
  else
    begin
      Pop := First^.Value; {Gán kết quả hàm}
      P := First^.Link; {Giữ lại nút tiếp theo First^ (Nút được đẩy vào danh sách ngay sau First^)}
```

```
      Dispose(First); First := P; {Giải phóng bộ nhớ cấp cho First^, cập nhật lại First mới}
    end;
end;

begin
  QueueInit;
  <Test>; {Đưa một vài lệnh để kiểm tra hoạt động của Queue}
end.
```

Bài tập

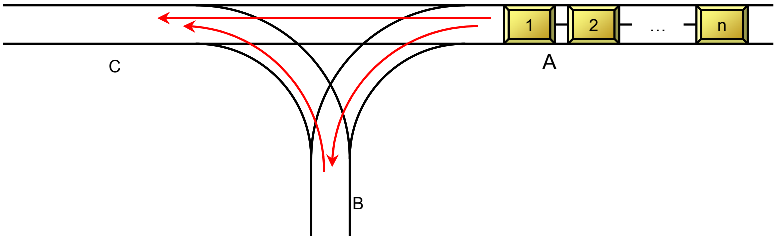
Bài 1.

Tìm hiểu cơ chế xếp chồng của thủ tục đệ quy, phương pháp dùng ngăn xếp để khử đệ quy.

Viết chương trình mô tả cách đổi cơ số từ hệ thập phân sang hệ cơ số R dùng ngăn xếp

Bài 3

Hình 13 là cơ cấu đường tàu tại một ga xe lửa



Hình 13: Di chuyển toa tàu

Ban đầu ở đường ray A chứa các toa tàu đánh số từ 1 tới n theo thứ tự từ trái qua phải, người ta muốn chuyển các toa đó sang đường ray C để được một thứ tự mới là một hoán vị của (1, 2, …, n) theo quy tắc: chỉ được đưa các toa tàu chạy theo đường ray theo hướng mũi tên, có thể dùng đoạn đường ray B để chứa tạm các toa tàu trong quá trình di chuyển.

a) Hãy nhập vào hoán vị cần có, cho biết có phương án chuyển hay không, và nếu có hãy đưa ra cách chuyển:

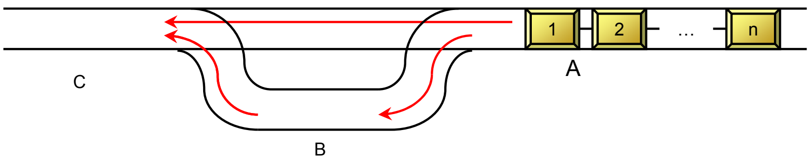
Ví dụ: n = 4; Thứ tự cần có (1, 4, 3, 2)

```
1)A → C; 2)A → B; 3)A → B; 4)A → C; 5)B → C; 6)B → C
```

b) Những hoán vị nào của thứ tự các toa là có thể tạo thành trên đoạn đường ray C với luật di chuyển như trên

Bài 4

Tương tự như bài 3, nhưng với sơ đồ đường ray sau:



Hình 14: Di chuyển toa tàu (2)

§6. CÂY (TREE)

6.1. ĐỊNH NGHĨA

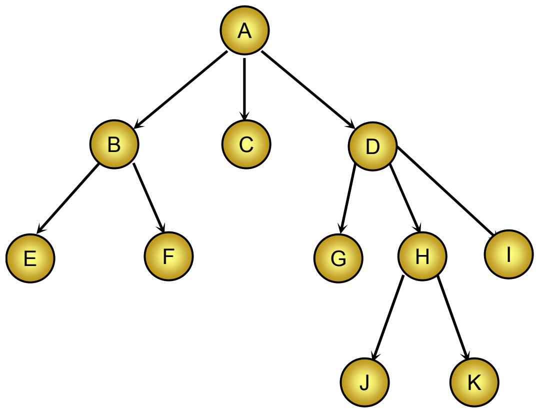
Cấu trúc dữ liệu trừu tượng ta quan tâm tới trong mục này là cấu trúc cây. Cây là một cấu trúc dữ liệu gồm một tập hữu hạn các nút, giữa các nút có một quan hệ phân cấp gọi là quan hệ "cha - con". Có một nút đặc biệt gọi là gốc (root).

Có thể định nghĩa cây bằng các đệ quy như sau:

- Mỗi nút là một cây, nút đó cũng là gốc của cây ấy
- Nếu n là một nút và n_1, n_2, …, n_k lần lượt là gốc của các cây T_1, T_2, …, T_k; các cây này đôi một không có nút chung. Thì nếu cho nút n trở thành cha của các nút n_1, n_2, …, n_k ta sẽ được một cây mới T. Cây này có nút n là gốc còn các cây T_1, T_2, …, T_k trở thành các cây con (subtree) của gốc.

Để tiện, người ta còn cho phép tồn tại một cây không có nút nào mà ta gọi là cây rỗng (null tree).

Xét cây trong Hình 15:



Hình 15: Cây

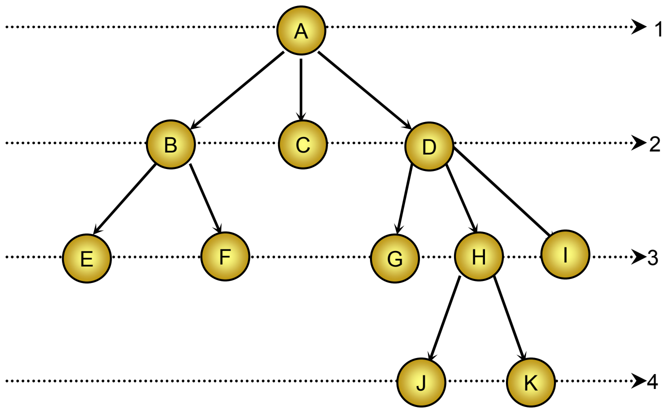
A là cha của B, C, D, còn G, H, I là con của D

Số các con của một nút được gọi là **cấp của nút** đó, ví dụ cấp của A là 3, cấp của B là 2, cấp của C là 0.

Nút có cấp bằng 0 được gọi là **nút lá** (leaf) hay nút tận cùng. Ví dụ như ở trên, các nút E, F, C, G, J, K và I là các nút là. Những nút không phải là lá được gọi là **nút nhánh** (branch)

Cấp cao nhất của một nút trên cây gọi là **cấp của cây** đó, cây ở hình trên là cây cấp 3.

Gốc của cây người ta gán cho số mức là 1, nếu nút cha có mức là i thì nút con sẽ có mức là i + 1. Mức của cây trong Hình 15 được chỉ ra trong Hình 16:



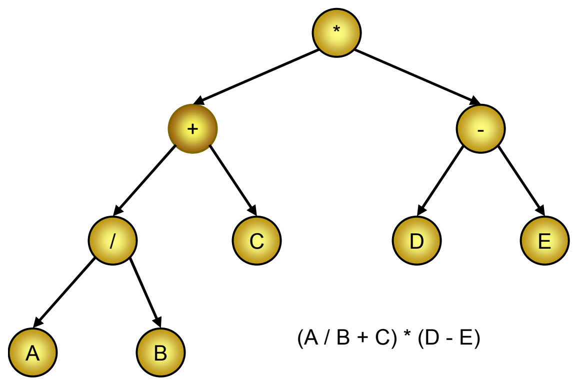
Hình 16: Mức của các nút trên cây

Chiều cao (height) hay **chiều sâu** (depth) của một cây là số mức lớn nhất của nút có trên cây đó. Cây ở trên có chiều cao là 4

Một tập hợp các cây phân biệt được gọi là **rừng** (forest), một cây cũng là một rừng. Nếu bỏ nút gốc trên cây thì sẽ tạo thành một rừng các cây con.

Ví dụ:

- Mục lục của một cuốn sách với phần, chương, bài, mục v.v… có cấu trúc của cây
- Cấu trúc thư mục trên đĩa cũng có cấu trúc cây, thư mục gốc có thể coi là gốc của cây đó với các cây con là các thư mục con và tệp nằm trên thư mục gốc.
- Gia phả của một họ tộc cũng có cấu trúc cây.
- Một biểu thức số học gồm các phép toán cộng, trừ, nhân, chia cũng có thể lưu trữ trong một cây mà các toán hạng được lưu trữ ở các nút lá, các toán tử được lưu trữ ở các nút nhánh, mỗi nhánh là một biểu thức con.



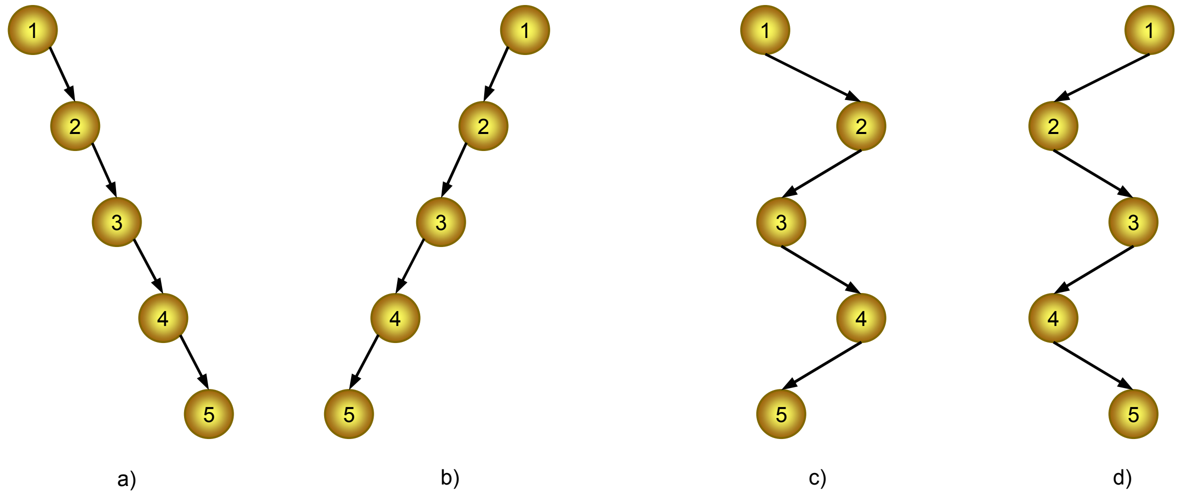
Hình 17: Cây biểu diễn biểu thức

6.2. CÂY NHỊ PHÂN (BINARY TREE)

Cây nhị phân là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là mọi nút trên cây chỉ có tối đa hai nhánh con. Với một nút thì người ta cũng phân biệt cây con trái và cây con phải của nút đó. Cây nhị phân là cây có tính đến thứ tự của các nhánh con.

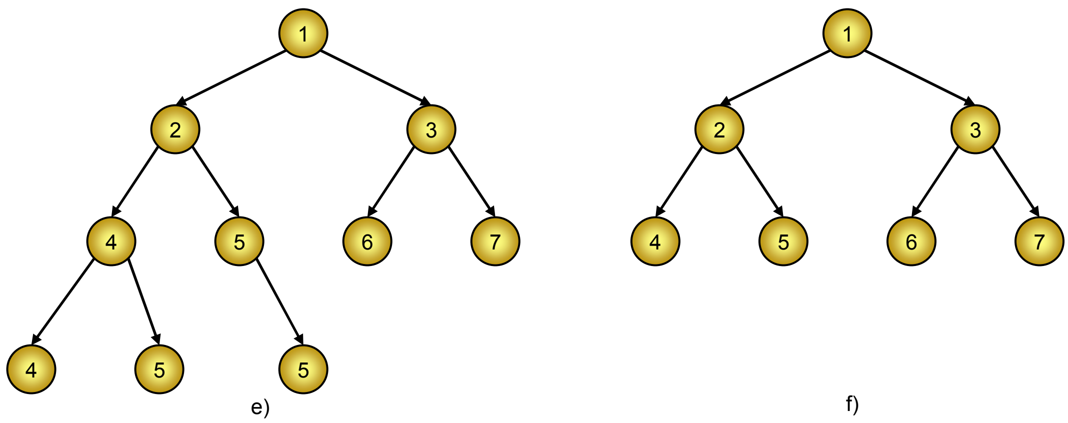
Cần chú ý tới một số dạng đặc biệt của cây nhị phân

Các cây nhị phân trong Hình 18**Error! Reference source not found.** được gọi là **cây nhị phân suy biến** (degenerate binary tree), các nút không phải là lá chỉ có một nhánh con. Cây a) được gọi là cây lệch phải, cây b) được gọi là cây lệch trái, cây c) và d) được gọi là cây zíc-zắc.



Hình 18: Các dạng cây nhị phân suy biến

Các cây trong Hình 19 được gọi là **cây nhị phân hoàn chỉnh** (complete binary tree): Nếu chiều cao của cây là h thì mọi nút có mức < h - 1 đều có đúng 2 nút con. Còn nếu mọi nút có mức ≤ h - 1 đều có đúng 2 nút con như trường hợp cây f) ở trên thì cây đó được gọi là **cây nhị phân đầy đủ** (full binary tree). Cây nhị phân đầy đủ là trường hợp riêng của cây nhị phân hoàn chỉnh.



Hình 19: Cây nhị phân hoàn chỉnh và cây nhị phân đầy đủ

Ta có thể thấy ngay những tính chất sau bằng phép chứng minh quy nạp:

Trong các cây nhị phân có cùng số lượng nút như nhau thì cây nhị phân suy biến có chiều cao lớn nhất, còn cây nhị phân hoàn chỉnh thì có chiều cao nhỏ nhất.

Số lượng tối đa các nút trên mức i của cây nhị phân là 2^{i-1}, tối thiểu là 1 (i ≥ 1).

Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là 2^h-1, tối thiểu là h (h ≥ 1).

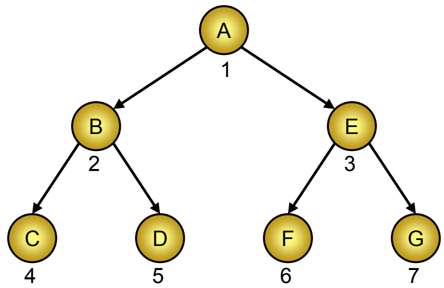
Cây nhị phân hoàn chỉnh, không đầy đủ, có n nút thì chiều cao của nó là $h = [\log_2(n + 1)] + 1$.

Cây nhị phân đầy đủ có n nút thì chiều cao của nó là $h = \log_2(n + 1)$

6.3. BIỂU DIỄN CÂY NHỊ PHÂN

6.3.1. Biểu diễn bằng mảng

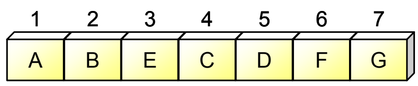
Nếu có một cây nhị phân đầy đủ, ta có thể dễ dàng đánh số cho các nút trên cây đó theo thứ tự lần lượt từ mức 1 trở đi, hết mức này đến mức khác và từ trái sang phải đối với các nút ở mỗi mức.



Hình 20: Đánh số các nút của cây nhị phân đầy đủ để biểu diễn bằng mảng

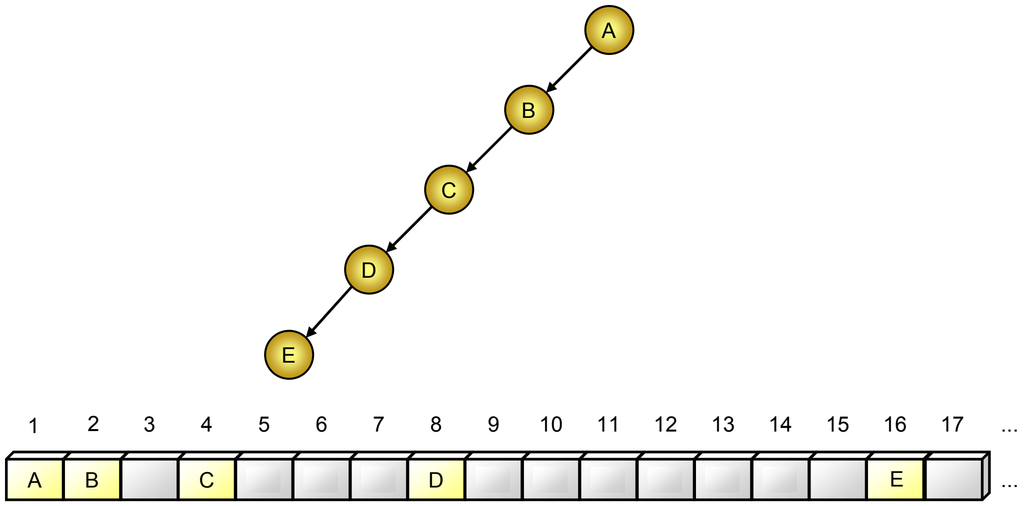
Với cách đánh số này, con của nút thứ i sẽ là các nút thứ 2i và 2i + 1. Cha của nút thứ j là nút j div 2. Từ đó có thể **lưu trữ cây bằng một mảng T, nút thứ i của cây được lưu trữ bằng phần tử T[i].**

Với cây nhị phân đầy đủ ở Hình 20 thì khi lưu trữ bằng mảng, ta sẽ được mảng như sau:



Trong trường hợp cây nhị phân không đầy đủ, ta có thể thêm vào một số nút giả để được cây nhị phân đầy đủ, và gán những giá trị đặc biệt cho những phần tử trong mảng T tương ứng với những nút này. Hoặc dùng thêm một mảng phụ để đánh dấu những nút nào là nút giả tự ta thêm vào. Chính vì lý do này nên với cây nhị phân không đầy đủ, ta sẽ gặp phải sự lãng phí bộ nhớ vì có thể sẽ phải thêm rất nhiều nút giả vào thì mới được cây nhị phân đầy đủ.

Ví dụ với cây lệch trái, ta phải dùng một mảng 31 phần tử để lưu cây nhị phân chỉ gồm 5 nút

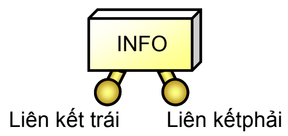


Hình 21: Nhược điểm của phương pháp biểu diễn cây bằng mảng

6.3.2. Biểu diễn bằng cấu trúc liên kết.

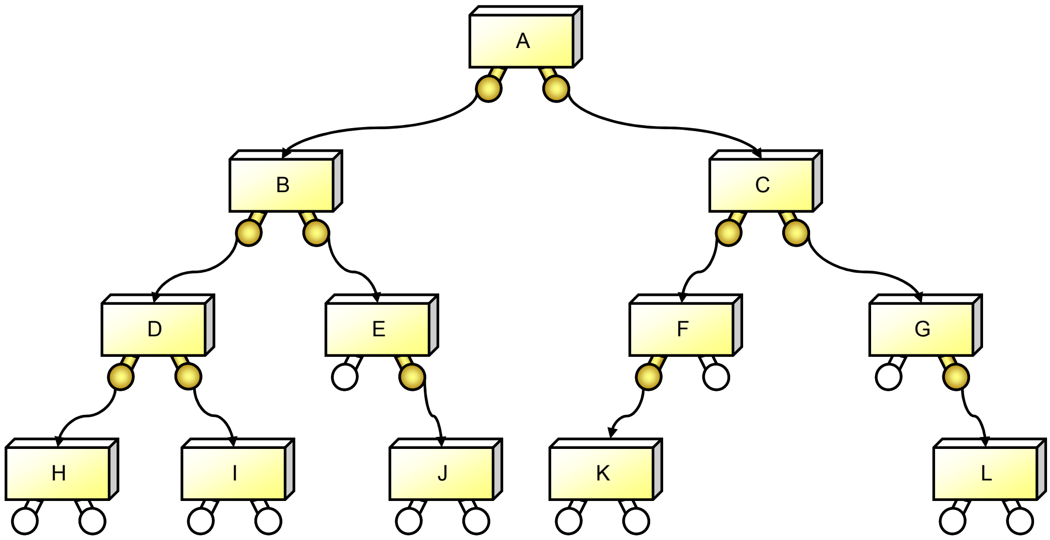
Khi biểu diễn cây nhị phân bằng cấu trúc liên kết, mỗi nút của cây là một bản ghi (record) gồm 3 trường:

- Trường Info: Chứa giá trị lưu tại nút đó
- Trường Left: Chứa liên kết (con trỏ) tới nút con trái, tức là chứa một thông tin đủ để biết nút con trái của nút đó là nút nào, trong trường hợp không có nút con trái, trường này được gán một giá trị đặc biệt.
- Trường Right: Chứa liên kết (con trỏ) tới nút con phải, tức là chứa một thông tin đủ để biết nút con phải của nút đó là nút nào, trong trường hợp không có nút con phải, trường này được gán một giá trị đặc biệt.



Hình 22: Cấu trúc nút của cây nhị phân

Đối với cây ta chỉ cần phải quan tâm giữ lại nút gốc, bởi từ nút gốc, đi theo các hướng liên kết Left, Right ta có thể duyệt mọi nút khác.



Hình 23: Biểu diễn cây bằng cấu trúc liên kết

6.4. PHÉP DUYỆT CÂY NHỊ PHÂN

Phép xử lý các nút trên cây mà ta gọi chung là phép thăm (Visit) các nút một cách hệ thống sao cho mỗi nút chỉ được thăm một lần gọi là phép duyệt cây.

Giả sử rằng nếu như một nút không có nút con trái (hoặc nút con phải) thì liên kết Left (Right) của nút đó được liên kết thẳng tới một nút đặc biệt mà ta gọi là NIL (hay NULL), nếu cây rỗng thì nút gốc của cây đó cũng được gán bằng NIL. Khi đó có ba cách duyệt cây hay được sử dụng:

6.4.1. Duyệt theo thứ tự trước (preorder traversal)

Trong phép duyệt theo thứ tự trước thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê trước giá trị lưu trong hai nút con của nó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(N); {Duyệt nhánh cây nhận N là nút gốc của nhánh đó}
begin
  if N ≠ nil then
    begin
      <Output trường Info của nút N>
      Visit(Nút con trái của N);
      Visit(Nút con phải của N);
    end;
end;
```

Quá trình duyệt theo thứ tự trước bắt đầu bằng lời gọi Visit(nút gốc).

Như cây ở Hình 23, nếu ta duyệt theo thứ tự trước thì các giá trị sẽ lần lượt được liệt kê theo thứ tự:

A B D H I E J C F K G L

6.4.2. Duyệt theo thứ tự giữa (inorder traversal)

Trong phép duyệt theo thứ tự giữa thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau giá trị lưu ở nút con trái và được liệt kê trước giá trị lưu ở nút con phải của nút đó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(N); {Duyệt nhánh cây nhận N là nút gốc của nhánh đó}
begin
  if N ≠ nil then
    begin
      Visit(Nút con trái của N);
      <Output trường Info của nút N>
      Visit(Nút con phải của N);
    end;
end;
```

Quá trình duyệt theo thứ tự giữa cũng bắt đầu bằng lời gọi Visit(nút gốc).

Như cây ở Hình 23, nếu ta duyệt theo thứ tự giữa thì các giá trị sẽ lần lượt được liệt kê theo thứ tự:

H D I B E J A K F C G L

6.4.3. Duyệt theo thứ tự sau (postorder traversal)

Trong phép duyệt theo thứ tự sau thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau giá trị lưu ở hai nút con của nút đó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(N); {Duyệt nhánh cây nhận N là nút gốc của nhánh đó}
begin
  if N ≠ nil then
    begin
      Visit(Nút con trái của N);
      Visit(Nút con phải của N);
      <Output trường Info của nút N>
    end;
end;
```

Quá trình duyệt theo thứ tự sau cũng bắt đầu bằng lời gọi Visit(nút gốc).

Cũng với cây ở Hình 23, nếu ta duyệt theo thứ tự sau thì các giá trị sẽ lần lượt được liệt kê theo thứ tự:

H I D J E B K F L G C A

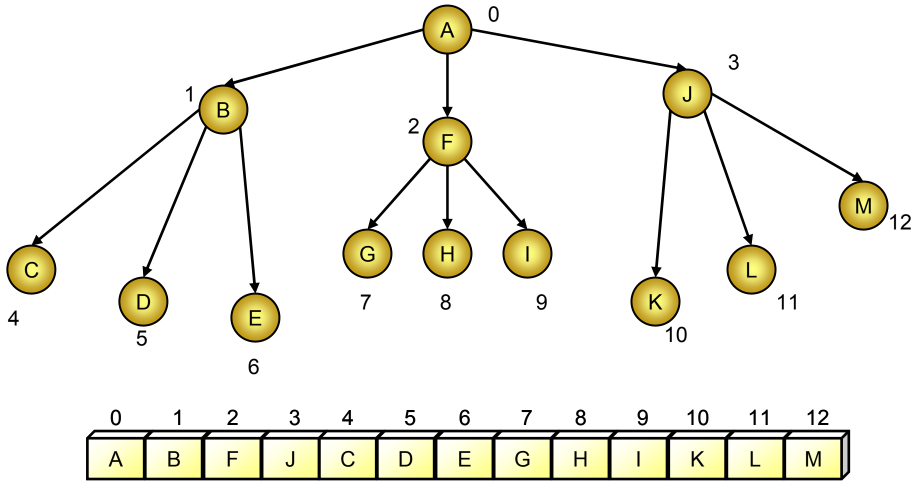
6.5. CÂY K_PHÂN

Cây K_phân là một dạng cấu trúc cây mà mỗi nút trên cây có tối đa K nút con (có tính đến thứ tự của các nút con).

6.5.1. Biểu diễn cây K_phân bằng mảng

Cũng tương tự như việc biểu diễn cây nhị phân, người ta có thể thêm vào cây K_phân một số nút giả để cho mỗi nút nhánh của cây K_phân đều có đúng K nút con, các nút con được xếp thứ tự từ nút con thứ nhất tới nút con thứ K, sau đó đánh số các nút trên cây K_phân bắt đầu từ 0 trở đi, bắt đầu từ mức 1, hết mức này đến mức khác và từ "trái qua phải" ở mỗi mức.

Theo cách đánh số này, nút con thứ j của nút i là: i * K + j. Nút cha của nút x là nút (x - 1) div K. Ta có thể dùng một mảng T đánh số từ 0 để lưu các giá trị trên các nút: Giá trị tại nút thứ i được lưu trữ ở phần tử T[i].



Hình 24: Đánh số các nút của cây 3_phân để biểu diễn bằng mảng

6.5.2. Biểu diễn cây K_phân bằng cấu trúc liên kết

Khi biểu diễn cây K_phân bằng cấu trúc liên kết, mỗi nút của cây là một bản ghi (record) gồm hai trường:

Trường Info: Chứa giá trị lưu trong nút đó.

Trường Links: Là một mảng gồm K phần tử, phần tử thứ i chứa liên kết (con trỏ) tới nút con thứ i, trong trường hợp không có nút con thứ i thì Links[i] được gán một giá trị đặc biệt.

Đối với cây K_ phân, ta cũng chỉ cần giữ lại nút gốc, bởi từ nút gốc, đi theo các hướng liên kết có thể đi tới mọi nút khác.

6.6. CÂY TỔNG QUÁT

Trong thực tế, có một số ứng dụng đòi hỏi một cấu trúc dữ liệu dạng cây nhưng không có ràng buộc gì về số con của một nút trên cây, ví dụ như cấu trúc thư mục trên đĩa hay hệ thống đề mục của một cuốn sách. Khi đó, ta phải tìm cách mô tả một cách khoa học cấu trúc dữ liệu dạng cây tổng quát. Cũng như trường hợp cây nhị phân, người ta thường biểu diễn cây tổng quát bằng hai cách: Lưu trữ kế tiếp bằng mảng và lưu trữ bằng cấu trúc liên kết.

6.6.1. Biểu diễn cây tổng quát bằng mảng

Để lưu trữ cây tổng quát bằng mảng, trước hết, ta đánh số các nút trên cây bắt đầu từ 1 theo một thứ tự tuỳ ý. Giả sử cây có n nút thì ta sử dụng:

Một mảng Info[1..n], trong đó Info[i] là giá trị lưu trong nút thứ i.

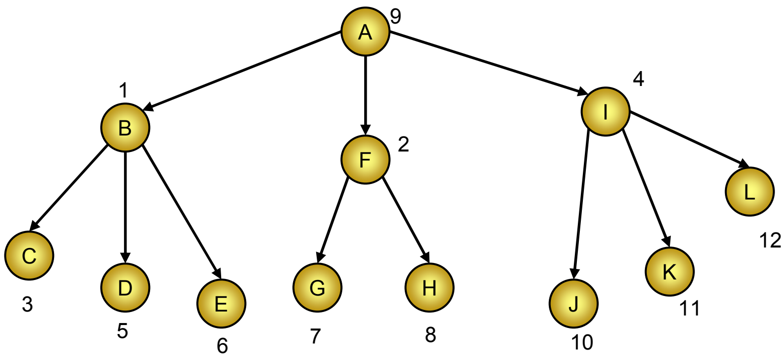
Một mảng Children được chia làm n đoạn, đoạn thứ i gồm một dãy liên tiếp các phần tử là chỉ số các nút con của nút i. Như vậy mảng Children sẽ chứa tất cả chỉ số của mọi nút con trên

cây (ngoại trừ nút gốc) nên nó sẽ gồm n - 1 phần tử, lưu ý rằng khi chia mảng Children làm n đoạn thì sẽ có những đoạn rỗng (tương ứng với danh sách các nút con của một nút lá)

Một mảng Head[1..n + 1], để đánh dấu vị trí cắt đoạn trong mảng Children: Head[i] là vị trí đứng liền trước đoạn thứ i, hay nói cách khác: Đoạn con tính từ chỉ số Head[i] + 1 đến Head[i] của mảng Children chứa chỉ số các nút con của nút thứ i. Khi Head[i] = Head[i+1] có nghĩa là đoạn thứ i rỗng. Quy ước: Head[n+1] = n - 1.

Giữ lại chỉ số của nút gốc.

Ví dụ:



	1	2	3	4	5	6	7	8	9	10	11	12	13
Info:	B	F	C	I	D	E	G	H	A	J	K	L	
Children:	3	5	6	7	8	10	11	12	1	2	4		
Head:	0	3	5	5	8	8	8	8	8	11	11	11	11

Hình 25: Biểu diễn cây tổng quát bằng mảng

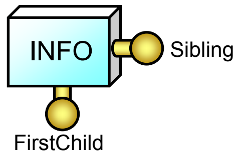
6.6.2. Lưu trữ cây tổng quát bằng cấu trúc liên kết

Khi lưu trữ cây tổng quát bằng cấu trúc liên kết, mỗi nút là một bản ghi (record) gồm ba trường:

Trường Info: Chứa giá trị lưu trong nút đó.

Trường FirstChild: Chứa liên kết (con trỏ) tới nút con đầu tiên của nút đó (con cả), trong trường hợp là nút lá (không có nút con), trường này được gán một giá trị đặc biệt.

Trường Sibling: Chứa liên kết (con trỏ) tới nút em kế cận bên phải (nút cùng cha với nút đang xét, khi sắp thứ tự các con thì nút đó đứng liền sau nút đang xét). Trong trường hợp không có nút em kế cận bên phải, trường này được gán một giá trị đặc biệt.



Hình 26: Cấu trúc nút của cây tổng quát

Dễ thấy được tính đúng đắn của phương pháp biểu diễn, bởi từ một nút N bất kỳ, ta có thể đi theo liên kết FirstChild để đến nút con cả, nút này chính là chốt của một danh sách nối đơn các nút con của nút N: từ nút con cả, đi theo liên kết Sibling, ta có thể duyệt tất cả các nút con của nút N.

Bài tập

Bài 1

Viết chương trình mô tả cây nhị phân dùng cấu trúc liên kết, mỗi nút chứa một số nguyên, và viết các thủ tục duyệt trước, giữa, sau.

Bài 2

Chứng minh rằng nếu cây nhị phân có x nút lá và y nút cấp 2 thì x = y + 1

Bài 3

Chứng minh rằng nếu ta biết dãy các nút được thăm của một cây nhị phân khi duyệt theo thứ tự trước và thứ tự giữa thì có thể dựng được cây nhị phân đó. Điều này con đúng nữa không đối với thứ tự trước và thứ tự sau? Với thứ tự giữa và thứ tự sau.

Bài 4

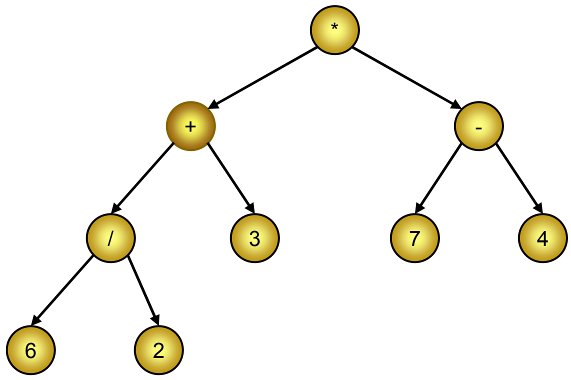
Viết các thủ tục duyệt trước, giữa, sau không đệ quy.

§7. KÝ PHÁP TIỀN TỐ, TRUNG TỐ VÀ HẬU TỐ

7.1. BIỂU THỨC DƯỚI DẠNG CÂY NHỊ PHÂN

Chúng ta có thể biểu diễn các biểu thức số học gồm các phép toán cộng, trừ, nhân, chia bằng một cây nhị phân, trong đó các nút lá biểu thị các hằng hay các biến (các toán hạng), các nút không phải là lá biểu thị các toán tử (phép toán số học chẳng hạn). Mỗi phép toán trong một nút sẽ tác động lên hai biểu thức con nằm ở cây con bên trái và cây con bên phải của nút đó.

Ví dụ: Cây biểu diễn biểu thức (6 / 2 + 3) * (7 - 4)



Hình 27: Biểu thức dưới dạng cây nhị phân

7.2. CÁC KÝ PHÁP CHO CÙNG MỘT BIỂU THỨC

Với cây nhị phân biểu diễn biểu thức trong Hình 27,

- Nếu duyệt theo thứ tự trước, ta sẽ được * + / 6 2 3 - 7 4, đây là **dạng tiền tố (prefix)** của biểu thức. Trong ký pháp này, toán tử được viết trước hai toán hạng tương ứng, người ta còn gọi ký pháp này là ký pháp Ba lan.
- Nếu duyệt theo thứ tự giữa, ta sẽ được 6 / 2 + 3 * 7 - 4. Ký pháp này hơi mập mờ vì thiếu dấu ngoặc. Nếu thêm vào thủ tục duyệt inorder việc bổ sung các cặp dấu ngoặc vào mỗi biểu thức con sẽ thu được biểu thức (((6 / 2) + 3) * (7 - 4)). Ký pháp này gọi là **dạng trung tố (infix)** của một biểu thức (Thực ra chỉ cần thêm các dấu ngoặc đủ để tránh sự mập mờ mà thôi, không nhất thiết phải thêm vào đầy đủ các cặp dấu ngoặc).
- Nếu duyệt theo thứ tự sau, ta sẽ được 6 2 / 3 + 7 4 - *, đây là **dạng hậu tố (postfix)** của biểu thức. Trong ký pháp này toán tử được viết sau hai toán hạng, người ta còn gọi ký pháp này là ký pháp nghịch đảo Balan (Reverse Polish Notation - RPN)

Chỉ có dạng trung tố mới cần có dấu ngoặc, dạng tiền tố và hậu tố không cần phải có dấu ngoặc.

7.3. CÁCH TÍNH GIÁ TRỊ BIỂU THỨC

Có một vấn đề cần lưu ý là khi máy tính giá trị một biểu thức số học gồm các toán tử hai ngôi (toán tử gồm hai toán hạng như +, -, *, /) thì máy chỉ thực hiện được phép toán đó với hai toán hạng. Nếu biểu thức phức tạp thì máy phải chia nhỏ và tính riêng từng biểu thức trung gian, sau đó mới lấy giá trị tìm được để tính tiếp. Ví dụ như biểu thức 1 + 2 + 4 máy sẽ phải tính 1 + 2 trước được kết quả là 3 sau đó mới đem 3 cộng với 4 chứ không thể thực hiện phép cộng một lúc ba số được.

Khi lưu trữ biểu thức dưới dạng cây nhị phân thì ta có thể coi **mỗi nhánh con của cây đó mô tả một biểu thức trung gian** mà máy cần tính khi xử lý biểu thức lớn. Như ví dụ trên, máy sẽ phải tính hai biểu thức 6 / 2 + 3 và 7 - 4 trước khi làm phép tính nhân cuối cùng. Để tính biểu thức 6 / 2 + 3 thì máy lại phải tính biểu thức 6 / 2 trước khi đem cộng với 3.

Vậy để tính một biểu thức lưu trữ trong một nhánh cây nhị phân gốc ở nút n, máy sẽ tính gần giống như hàm đệ quy sau:

```
function Calculate(n): Value; {Tính biểu thức con trong nhánh cây gốc n}
begin
 if <Nút n chứa không phải là một toán tử> then
   Calculate := <Giá trị chứa trong nút n>
 else {Nút n chứa một toán tử R}
   begin
     x := Calculate(nút con trái của n);
     y := Calculate(nút con phải của n);
     Calculate := x R y;
   end;
end.
```

(Trong trường hợp lập trình trên các hệ thống song song, việc tính giá trị biểu thức ở cây con trái và cây con phải có thể tiến hành đồng thời làm giảm đáng kể thời gian tính toán biểu thức).

Để ý rằng khi tính toán biểu thức, máy sẽ phải quan tâm tới việc tính biểu thức ở hai nhánh con trước, rồi mới xét đến toán tử ở nút gốc. Điều đó làm ta nghĩ tới phép cây theo thứ tự sau và ký pháp hậu tố. Trong những năm đầu 1950, nhà lô-gic học người Balan Jan Lukasiewicz đã chứng minh rằng biểu thức hậu tố không cần phải có dấu ngoặc vẫn có thể tính được một cách đúng đắn bằng cách **đọc lần lượt biểu thức từ trái qua phải** và dùng một Stack để lưu các kết quả trung gian:

Bước 1: Khởi động một Stack rỗng

Bước 2: Đọc lần lượt các phần tử của biểu thức RPN từ trái qua phải (phần tử này có thể là hằng, biến hay toán tử) với mỗi phần tử đó, ta kiểm tra:

Nếu phần tử này là một toán hạng thì đẩy giá trị của nó vào Stack.

Nếu phần tử này là một toán tử ®, ta lấy từ Stack ra hai giá trị (y và x) sau đó áp dụng toán tử ® đó vào hai giá trị vừa lấy ra, đẩy kết quả tìm được (x ® y) vào Stack (ra hai vào một).

Bước 3: Sau khi kết thúc bước 2 thì toàn bộ biểu thức đã được đọc xong, trong Stack chỉ còn duy nhất một phần tử, phần tử đó chính là giá trị của biểu thức.

Ví dụ: Tính biểu thức 10 2 / 3 + 7 4 - * (tương ứng với biểu thức (10 / 2 + 3) * (7 - 4)

Đọc	Xử lý	Stack
10	Đẩy vào Stack	10
2	Đẩy vào Stack	10, 2
/	Lấy 2 và 10 khỏi Stack, Tính được 10 / 2 = 5, đẩy 5 vào Stack	5
3	Đẩy vào Stack	5, 3
+	Lấy 3 và 5 khỏi Stack, tính được 5 + 3 = 8, đẩy 8 vào Stack	8
7	Đẩy vào Stack	8, 7
4	Đẩy vào Stack	8, 7, 4
-	Lấy 4 và 7 khỏi Stack, tính được 7 - 4 = 3, đẩy 3 vào Stack	8, 3
*	Lấy 3 và 8 khỏi Stack, tính được 8 * 3 = 24, đẩy 24 vào Stack	24

Ta được kết quả là 24

Dưới đây ta sẽ viết một chương trình đơn giản tính giá trị biểu thức RPN. Chương trình sẽ nhận Input là biểu thức RPN gồm các số thực và các toán tử + - * / và cho Output là kết quả biểu thức đó.

Quy định khuôn dạng bắt buộc là hai số liền nhau trong biểu thức RPN phải viết cách nhau ít nhất một dấu cách. Để quá trình đọc một phần tử trong biểu thức RPN được dễ dàng hơn, sau bước nhập liệu, ta có thể hiệu chỉnh đôi chút biểu thức RPN về khuôn dạng dễ đọc nhất. Chẳng hạn như thêm và bớt một số dấu cách trong Input để mỗi phần tử (toán hạng, toán tử) đều cách nhau đúng một dấu cách, thêm một dấu cách vào cuối biểu thức RPN. Khi đó quá trình đọc lần lượt các phần tử trong biểu thức RPN có thể làm như sau:

```
T := '';
for p := 1 to Length(RPN) do {Xét các ký tự trong biểu thức RPN từ trái qua phải}
  if RPN[p] ≠ ' ' then T := T + RPN[p] {Nếu RPN[p] không phải dấu cách thì nối ký tự đó vào T}
  else {Nếu RPN[p] là dấu cách thì phần tử đang đọc đã đọc xong, tiếp theo sẽ là phần tử khác}
    begin
      <Xử lý phần tử T>
      T := ''; {Chuẩn bị đọc phần tử mới}
    end;
```

Để đơn giản, chương trình không kiểm tra lỗi viết sai biểu thức RPN, việc đó chỉ là thao tác tỉ mỉ chứ không phức tạp lắm, chỉ cần xem lại thuật toán và cài thêm các mô-đun bắt lỗi tại mỗi bước.

Ví dụ về Input / Output của chương trình:

```
Enter RPN Expression: 10 2/3    +   4 7 -*
10 2 / 3 + 4 7 - *  = 24.0000
```

P_2_07_1.PAS * Tính giá trị biểu thức RPN

```
{$N+,E+}
program CalculateRPNExpression;
const
  Opt = ['+', '-', '*', '/'];
```

```
var
  T, RPN: String;
  Stack: array[1..255] of Extended;
  p, Last: Integer;
{Các thao tác đối với Stack}
procedure StackInit;
begin
  Last := 0;
end;

procedure Push(V: Extended);
begin
  Inc(Last); Stack[Last] := V;
end;

function Pop: Extended;
begin
  Pop := Stack[Last]; Dec(Last);
end;

procedure Refine(var S: String); {Hiệu chỉnh biểu thức RPN về khuôn dạng dễ đọc nhất}
var
  i: Integer;
begin
  S := S + ' ';
  for i := Length(S) - 1 downto 1 do {Thêm những dấu cách giữa toán hạng và toán tử}
    if (S[i] in Opt) or (S[i + 1] in Opt) then
      Insert(' ', S, i + 1);
  for i := Length(S) - 1 downto 1 do {Xoá những dấu cách thừa}
    if (S[i] = ' ') and (S[i + 1] = ' ') then Delete(S, i + 1, 1);
end;

procedure Process(T: String); {Xử lý phần tử T đọc được từ biểu thức RPN}
var
  x, y: Extended;
  e: Integer;
begin
  if not (T[1] in Opt) then {T là toán hạng}
    begin
      Val(T, x, e); Push(x); {Đổi T thành số và đẩy giá trị đó vào Stack}
    end
  else {T là toán tử}
    begin
      y := Pop; x := Pop; {Ra hai}
      case T[1] of
        '+': x := x + y;
        '-': x := x - y;
        '*': x := x * y;
        '/': x := x / y;
      end;
      Push(x); {Vào một}
    end;
end;

begin
  Write('Enter RPN Expression: '); ReadLn(RPN);
  Refine(RPN);
  StackInit;
  T := '';
  for p := 1 to Length(RPN) do {Xét các ký tự của biểu thức RPN từ trái qua phải}
    if RPN[p] <> ' ' then T := T + RPN[p] {nếu không phải dấu cách thì nối nó vào sau xâu T}
    else {Nếu gặp dấu cách}
      begin
        Process(T); {Xử lý phần tử vừa đọc xong}
```

```
      T := ''; {Đặt lại T để chuẩn bị đọc phần tử mới}
    end;
  WriteLn(RPN, ' = ', Pop:0:4); {In giá trị biểu thức RPN được lưu trong Stack}
end.
```

7.4. CHUYỂN TỪ DẠNG TRUNG TỐ SANG DẠNG HẬU TỐ

Có thể nói rằng việc tính toán biểu thức viết bằng ký pháp nghịch đảo Balan là khoa học hơn, máy móc, và đơn giản hơn việc tính toán biểu thức viết bằng ký pháp trung tố. Chỉ riêng việc không phải xử lý dấu ngoặc đã cho ta thấy ưu điểm của ký pháp RPN. Chính vì lý do này, các chương trình dịch vẫn cho phép lập trình viên viết biểu thức trên ký pháp trung tố theo thói quen, nhưng trước khi dịch ra các lệnh máy thì tất cả các biểu thức đều được chuyển về dạng RPN. Vấn đề đặt ra là phải có một thuật toán chuyển biểu thức dưới dạng trung tố về dạng RPN một cách hiệu quả, và dưới đây ta trình bày thuật toán đó:

Thuật toán sử dụng một Stack để chứa các toán tử và dấu ngoặc mở. Thủ tục Push(V) để đẩy một phần tử vào Stack, hàm Pop để lấy ra một phần tử từ Stack, hàm Get để đọc giá trị phần tử nằm ở đỉnh Stack mà không lấy phần tử đó ra. Ngoài ra mức độ ưu tiên của các toán tử được quy định bằng hàm Priority như sau: Ưu tiên cao nhất là dấu "*" và "/" với Priority là 2, tiếp theo là dấu "+" và "-" với Priority là 1, ưu tiên thấp nhất là dấu ngoặc mở "(" với Priority là 0.

```
Stack := Ø;
for <Phần tử T đọc được từ biểu thức infix> do
{T có thể là hằng, biến, toán tử hoặc dấu ngoặc được đọc từ biểu thức infix theo thứ tự từ trái qua phải}
  case T of
    '(': Push(T);
    ')':
         repeat
           x := Pop;
           if x ≠ '(' then Output(x);
         until x = '(';
    '+', '-', '*', '/':
         begin
           while (Stack ≠ Ø) and (Priority(T) ≤ Priority(Get)) do Output(Pop);
           Push(T);
         end;
    else Output(T);
  end;
while (Stack ≠ Ø) do Output(Pop);
```

Ví dụ với biểu thức trung tố (2 * 3 + 7 / 8) * (5 - 1)

Đọc	Xử lý	Stack	Output
(	Đẩy vào Stack	(	
2	Hiển thị	(	2
*	phép "*" được ưu tiên hơn phần tử ở đỉnh Stack là "(", đẩy "*" vào Stack	(*	
3	Hiển thị	(*	2 3
+	phép "+" ưu tiên không cao hơn phần tử ở đỉnh Stack là "*", lấy ra và hiển thị "*". So sánh tiếp, thấy phép "+" được ưu tiên cao hơn phần tử ở đỉnh Stack là "(", đẩy "+" vào Stack	(+	2 3 *
7	Hiển thị	(+	2 3 * 7

Đọc	Xử lý	Stack	Output
/	phép "/" được ưu tiên hơn phần tử ở đỉnh Stack là "+", đẩy "/" vào Stack	(+/	
8	Hiển thị	(+/	2 3 * 7 8
)	Lấy ra và hiển thị các phần tử trong Stack tới khi lấy phải dấu ngoặc mở	∅	2 3 * 7 8 / +
*	Stack đang là rỗng, đẩy * vào Stack	*	
(	Đẩy vào Stack	*(	
5	Hiển thị	*(	2 3 * 7 8 / + 5
-	phép "-" được ưu tiên hơn phần tử ở đỉnh Stack là "(", đẩy "-" vào Stack	*(-	
1	Hiển thị	*(-	2 3 * 7 8 / + 5 1
)	Lấy ra và hiển thị các phần tử ở đỉnh Stack cho tới khi lấy phải dấu ngoặc mở	*	2 3 * 7 8 / + 5 1 -
Hết	Lấy ra và hiển thị hết các phần tử còn lại trong Stack		2 3 * 7 8 / + 5 1 - *

Dưới đây là chương trình chuyển biểu thức viết ở dạng trung tố sang dạng RPN. Biểu thức trung tố đầu vào sẽ được hiệu chỉnh sao cho mỗi thành phần của nó được cách nhau đúng một dấu cách, và thêm một dấu cách vào cuối cho dễ tách các phần tử ra để xử lý. Vì Stack chỉ dùng để chứa các toán tử và dấu ngoặc mở nên có thể mô tả Stack dưới dạng xâu ký tự cho đơn giản.

Ví dụ về Input / Output của chương trình:

```
Infix:   (10*3 +    7 /8) * (5-1)
Refined: ( 10 * 3 + 7 / 8 ) * ( 5 - 1 )
RPN: 10 3 * 7 8 / + 5 1 - *
```

P_2_07_2.PAS * Chuyển biểu thức trung tố sang dạng RPN

```
program ConvertInfixToRPN;
const
  Opt = ['(', ')', '+', '-', '*', '/'];
var
  T, Infix, Stack: String; {Stack dùng để chứa toán tử và dấu ngoặc mở nên dùng String cho tiện}
  p: Integer;

{Các thao tác đối với Stack}
procedure StackInit;
begin
  Stack := '';
end;

procedure Push(V: Char);
begin
  Stack := Stack + V;
end;

function Pop: Char;
begin
  Pop := Stack[Length(Stack)];
  Dec(Stack[0]);
end;

function Get: Char;
begin
  Get := Stack[Length(Stack)];
end;

procedure Refine(var S: String); {Hiệu chỉnh biểu thức trung tố về khuôn dạng dễ đọc nhất}
var
  i: Integer;
begin
```

```
  S := S + ' ';
  for i := Length(S) - 1 downto 1 do {Thêm những dấu cách trước và sau mỗi toán tử và dấu ngoặc}
    if (S[i] in Opt) or (S[i + 1] in Opt) then
      Insert(' ', S, i + 1);
  for i := Length(S) - 1 downto 1 do {Xoá những dấu cách thừa}
    if (S[i] = ' ') and (S[i + 1] = ' ') then Delete(S, i + 1, 1);
end;

function Priority(Ch: Char): Integer; {Hàm lấy mức độ ưu tiên của Ch}
begin
  case ch of
    '*', '/': Priority := 2;
    '+', '-': Priority := 1;
    '(': Priority := 0;
  end;
end;

procedure Process(T: String); {Xử lý một phần tử đọc được từ biểu thức trung tố}
var
  c, x: Char;
begin
  c := T[1];
  if not (c in Opt) then Write(T, ' ')
  else
    case c of
      '(': Push(c);
      ')': repeat
             x := Pop;
             if x <> '(' then Write(x, ' ');
           until x = '(';
      '+', '-', '*', '/':
           begin
             while (Stack <> '') and (Priority(c) <= Priority(Get)) do
               Write(Pop, ' ');
             Push(c);
           end;
    end;
end;

begin
  Write('Infix = '); ReadLn(Infix);
  Refine(Infix);
  WriteLn('Refined: ', Infix);
  Write('RPN: ');
  T := '';
  for p := 1 to Length(Infix) do
    if Infix[p] <> ' ' then T := T + Infix[p]
    else
      begin
        Process(T);
        T := '';
      end;
  while Stack <> '' do Write(Pop, ' ');
  WriteLn;
end.
```

7.5. XÂY DỰNG CÂY NHỊ PHÂN BIỂU DIỄN BIỂU THỨC

Ngay trong phần đầu tiên, chúng ta đã biết rằng các dạng biểu thức trung tố, tiền tố và hậu tố đều có thể được hình thành bằng cách duyệt cây nhị phân biểu diễn biểu thức đó theo các trật tự khác nhau. Vậy tại sao không xây dựng ngay cây nhị phân biểu diễn biểu thức đó rồi thực

hiện các công việc tính toán ngay trên cây?. Khó khăn gặp phải chính là thuật toán xây dựng cây nhị phân trực tiếp từ dạng trung tố có thể kém hiệu quả, trong khi đó từ dạng hậu tố lại có thể khôi phục lại cây nhị phân biểu diễn biểu thức một cách rất đơn giản, gần giống như quá trình tính toán biểu thức hậu tố:

Bước 1: Khởi tạo một Stack rỗng dùng để chứa các nút trên cây

Bước 2: Đọc lần lượt các phần tử của biểu thức RPN từ trái qua phải (phần tử này có thể là hằng, biến hay toán tử) với mỗi phần tử đó:

Tạo ra một nút mới N chứa phần tử mới đọc được

Nếu phần tử này là một toán tử, lấy từ Stack ra hai nút (theo thứ tự là y và x), sau đó đem liên kết trái của N trỏ đến x, đem liên kết phải của N trỏ đến y.

Đẩy nút N vào Stack

Bước 3: Sau khi kết thúc bước 2 thì toàn bộ biểu thức đã được đọc xong, trong Stack chỉ còn duy nhất một phần tử, phần tử đó chính là gốc của cây nhị phân biểu diễn biểu thức.

Bài tập

Bài 1

Viết chương trình chuyển biểu thức trung tố sang dạng RPN, biểu thức trung tố có cả những phép toán một ngôi: Phép lấy số đối (-x), phép luỹ thừa x^y (x^y), lời gọi hàm số học (sqrt, exp, abs v.v…)

Bài 2

Viết chương trình chuyển biểu thức logic dạng trung tố sang dạng RPN. Ví dụ:

Chuyển: a and b or c and d thành: a b and c d and or

Bài 3

Chuyển các biểu thức sau đây ra dạng RPN

a) A * (B + C) b) A + B / C + D
c) A * (B + -C)   d) A - $(B + C)^{d/e}$
e) A and B or C   f) A and (B or not C)
g) (A or B) and (C or (D and not E))   h) (A = B) or (C = D)
i) (A < 9) and (A > 3) or not (A > 0)
j) ((A > 0) or (A < 0)) and (B * B - 4 * A * C < 0)

Bài 4

Viết chương trình tính biểu thức logic dạng RPN với các toán tử and, or, not và các toán hạng là TRUE hay FALSE.

Bài 5

Viết chương trình hoàn chỉnh tính giá trị biểu thức trung tố.

§8. SẮP XẾP (SORTING)

8.1. BÀI TOÁN SẮP XẾP

Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự nhất định. Chẳng hạn như thứ tự tăng dần (hay giảm dần) đối với một dãy số, thứ tự từ điển đối với các từ v.v… Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng Tin học với các mục đích khác nhau: sắp xếp dữ liệu trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu v.v…

Nói chung, dữ liệu có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước: Một tập các đối tượng cần sắp xếp là tập các bản ghi (records), mỗi bản ghi bao gồm một số trường (fields) khác nhau. Nhưng không phải toàn bộ các trường dữ liệu trong bản ghi đều được xem xét đến trong quá trình sắp xếp mà chỉ là một trường nào đó (hay một vài trường nào đó) được chú ý tới thôi. Trường như vậy ta gọi là **khoá (key)**. Sắp xếp sẽ được tiến hành dựa vào giá trị của khoá này.

Ví dụ: Hồ sơ tuyển sinh của một trường Đại học là một danh sách thí sinh, mỗi thí sinh có tên, số báo danh, điểm thi. Khi muốn liệt kê danh sách những thí sinh trúng tuyển tức là phải sắp xếp các thí sinh theo thứ tự từ điểm cao nhất tới điểm thấp nhất. Ở đây khoá sắp xếp chính là điểm thi.

STT	*SBD*	*Họ và tên*	*Điểm thi*
1	*A100*	*Nguyễn Văn A*	*20*
2	*B200*	*Trần Thị B*	*25*
3	*X150*	*Phạm Văn C*	*18*
4	*G180*	*Đỗ Thị D*	*21*

Khi sắp xếp, các bản ghi trong bảng sẽ được đặt lại vào các vị trí sao cho giá trị khoá tương ứng với chúng có đúng thứ tự đã ấn định. Vì kích thước của toàn bản ghi có thể rất lớn, nên nếu việc sắp xếp thực hiện trực tiếp trên các bản ghi sẽ đòi hỏi sự chuyển đổi vị trí của các bản ghi, kéo theo việc thường xuyên phải di chuyển, copy những vùng nhớ lớn, gây ra những tổn phí thời gian khá nhiều. Thường người ta khắc phục tình trạng này bằng cách xây dựng một bảng khoá: Mỗi bản ghi trong bảng ban đầu sẽ tương ứng với một bản ghi trong **bảng khoá**. Bảng khoá cũng gồm các bản ghi nhưng mỗi bản ghi chỉ gồm có hai trường:

Trường thứ nhất chứa khoá

Trường thứ hai chứa liên kết tới một bản ghi trong bảng ban đầu, tức là chứa một thông tin đủ để biết bản ghi tương ứng với nó trong bảng ban đầu là bản ghi nào.

Sau đó, việc sắp xếp được thực hiện trực tiếp trên bảng khoá, trong quá trình sắp xếp, **bảng chính không hề bị ảnh hưởng gì**, việc truy cập vào một bản ghi nào đó của bảng chính vẫn

có thể thực hiện được bằng cách dựa vào trường liên kết của bản ghi tương ứng thuộc bảng khoá.

Như ở ví dụ trên, ta có thể xây dựng bảng khoá gồm 2 trường, trường khoá chứa điểm và trường liên kết chứa số thứ tự của người có điểm tương ứng trong bảng ban đầu:

Điểm thi	STT
20	1
25	2
18	3
21	4

Sau khi sắp xếp theo trật tự điểm cao nhất tới điểm thấp nhất, bảng khoá sẽ trở thành:

Điểm thi	STT
25	2
21	4
20	1
18	3

Dựa vào bảng khoá, ta có thể biết được rằng người có điểm cao nhất là người mang số thứ tự 2, tiếp theo là người mang số thứ tự 4, tiếp nữa là người mang số thứ tự 1, và cuối cùng là người mang số thứ tự 3, còn muốn liệt kê danh sách đầy đủ thì ta chỉ việc đối chiếu với bảng ban đầu và liệt kê theo thứ tự 2, 4, 1, 3.

Có thể còn cải tiến tốt hơn dựa vào nhận xét sau: Trong bảng khoá, nội dung của trường khoá hoàn toàn có thể suy ra được từ trường liên kết bằng cách: Dựa vào trường liên kết, tìm tới bản ghi tương ứng trong bảng chính rồi truy xuất trường khoá trong bảng chính. Như ví dụ trên thì người mang số thứ tự 1 chắc chắn sẽ phải có điểm thi là 20, còn người mang số thứ tự 3 thì chắc chắn phải có điểm thi là 18. Vậy thì bảng khoá có thể loại bỏ đi trường khoá mà chỉ giữ lại trường liên kết. Trong trường hợp các phần tử trong bảng ban đầu được đánh số từ 1 tới n và trường liên kết chính là số thứ tự của bản ghi trong bảng ban đầu như ở ví dụ trên, người ta gọi kỹ thuật này là kỹ thuật **sắp xếp bằng chỉ số**: Bảng ban đầu không hề bị ảnh hưởng gì cả, việc sắp xếp chỉ đơn thuần là đánh lại chỉ số cho các bản ghi theo thứ tự sắp xếp. Cụ thể hơn:

Nếu r[1], r[2], …, r[n] là các bản ghi cần sắp xếp theo một thứ tự nhất định thì việc sắp xếp bằng chỉ số tức là xây dựng một dãy Index[1], Index[2], …, Index[n] mà ở đây:

Index[j] = Chỉ số của bản ghi sẽ đứng thứ j khi sắp thứ tự

(Bản ghi r[index[j]] sẽ phải đứng sau j - 1 bản ghi khác khi sắp xếp)

Do khoá có vai trò đặc biệt như vậy nên sau này, khi trình bày các giải thuật, ta sẽ coi **khoá như đại diện cho các bản ghi** và để cho đơn giản, ta chỉ nói tới giá trị của khoá mà thôi. Các thao tác trong kỹ thuật sắp xếp lẽ ra là tác động lên toàn bản ghi giờ đây chỉ làm trên khoá.

Còn việc cài đặt các phương pháp sắp xếp trên danh sách các bản ghi và kỹ thuật sắp xếp bằng chỉ số, ta coi như bài tập.

Bài toán sắp xếp giờ đây có thể phát biểu như sau:

Xét quan hệ thứ tự toàn phần "nhỏ hơn hoặc bằng" ký hiệu "≤" trên một tập hợp S, là quan hệ hai ngôi thoả mãn bốn tính chất:

Với $\forall a, b, c \in S$

Tính phổ biến: Hoặc là $a \le b$, hoặc $b \le a$;

Tính phản xạ: $a \le a$

Tính phản đối xứng: Nếu $a \le b$ và $b \le a$ thì bắt buộc $a = b$.

Tính bắc cầu: Nếu có $a \le b$ và $b \le c$ thì $a \le c$.

Trong trường hợp $a \le b$ và $a \neq b$, ta dùng ký hiệu "<" cho gọn

Cho một dãy gồm n khoá. Giữa hai khoá bất kỳ có quan hệ thứ tự toàn phần "≤". Xếp lại dãy các khoá đó để được dãy khoá thoả mãn $k_1 \le k_2 \le \ldots \le k_n$.

Giả sử cấu trúc dữ liệu cho dãy khoá được mô tả như sau:

```
const
  n = …; {Số khoá trong dãy khoá, có thể khai dưới dạng biến số nguyên để tuỳ biến hơn}
type
  TKey = …; {Kiểu dữ liệu một khoá}
  TArray = array[1..n] of TKey;
var
  k: TArray; {Dãy khoá}
```

Thì những thuật toán sắp xếp dưới đây được viết dưới dạng thủ tục sắp xếp dãy khoá k, kiểu chỉ số đánh cho từng khoá trong dãy có thể coi là số nguyên Integer.

8.2. THUẬT TOÁN SẮP XẾP KIỂU CHỌN (SELECTIONSORT)

Một trong những thuật toán sắp xếp đơn giản nhất là phương pháp sắp xếp kiểu chọn. Ý tưởng cơ bản của cách sắp xếp này là:

Ở lượt thứ nhất, ta chọn trong dãy khoá $k_1, k_2, \ldots, k_n$ ra khoá nhỏ nhất (khoá ≤ mọi khoá khác) và đổi giá trị của nó với k_1, khi đó giá trị khoá k_1 trở thành giá trị khoá nhỏ nhất.

Ở lượt thứ hai, ta chọn trong dãy khoá $k_2, \ldots, k_n$ ra khoá nhỏ nhất và đổi giá trị của nó với k_2.

…

Ở lượt thứ i, ta chọn trong dãy khoá $k_i, k_{i+1}, \ldots$, kn ra khoá nhỏ nhất và đổi giá trị của nó với k_i.

…

Làm tới lượt thứ n - 1, chọn trong hai khoá k_{n-1}, k_n ra khoá nhỏ nhất và đổi giá trị của nó với k_{n-1}.

```
procedure SelectionSort;
var
  i, j, jmin: Integer;
begin
  for i := 1 to n - 1 do {Làm n - 1 lượt}
    begin
      {Chọn trong số các khoá từ ki tới kn ra khoá kjmin nhỏ nhất}
      jmin := i;
      for j := i + 1 to n do
        if kj < kjmin then jmin := j;
      if jmin ≠ i then
        <Đảo giá trị của kjmin cho ki>
    end;
end;
```

Đối với phương pháp kiểu lựa chọn, ta có thể coi phép so sánh ($k_j < k_{jmin}$) là phép toán tích cực để đánh giá hiệu suất thuật toán về mặt thời gian. Ở lượt thứ i, để chọn ra khoá nhỏ nhất bao giờ cũng cần n - i phép so sánh, số lượng phép so sánh này không hề phụ thuộc gì vào tình trạng ban đầu của dãy khoá cả. Từ đó suy ra tổng số phép so sánh sẽ phải thực hiện là:

$$(n - 1) + (n - 2) + \ldots + 1 = n * (n - 1) / 2$$

Vậy **thuật toán sắp xếp kiểu chọn có cấp là $O(n^2)$**

8.3. THUẬT TOÁN SẮP XẾP NỔI BỌT (BUBBLESORT)

Trong thuật toán sắp xếp nổi bọt, dãy các khoá sẽ được duyệt từ cuối dãy lên đầu dãy (từ k_n về k_1), nếu gặp hai khoá kế cận bị ngược thứ tự thì đổi chỗ của chúng cho nhau. Sau lần duyệt như vậy, phần tử nhỏ nhất trong dãy khoá sẽ được chuyển về vị trí đầu tiên và vấn đề trở thành sắp xếp dãy khoá từ k_2 tới k_n:

```
procedure BubbleSort;
var
  i, j: Integer;
begin
  for i := 2 to n do
    for j := n downto i do {Duyệt từ cuối dãy lên, làm nổi khoá nhỏ nhất trong số ki-1, …,kn về vị trí i-1}
      if kj < kj-1 then
        <Đảo giá trị kj và kj-1>
end;
```

Đối với thuật toán sắp xếp nổi bọt, ta có thể coi phép toán tích cực là phép so sánh $k_j < k_{j-1}$. Và số lần thực hiện phép so sánh này là:

$$(n - 1) + (n - 2) + \ldots + 1 = n * (n - 1) / 2$$

Vậy **thuật toán sắp xếp nổi bọt cũng có cấp là $O(n^2)$.** Bất kể tình trạng dữ liệu vào như thế nào.

8.4. THUẬT TOÁN SẮP XẾP KIỂU CHÈN

Xét dãy khoá $k_1, k_2, \ldots, k_n$. Ta thấy dãy con chỉ gồm mỗi một khoá là k_1 có thể coi là đã sắp xếp rồi. Xét thêm k_2, ta so sánh nó với k_1, nếu thấy $k_2 < k_1$ thì chèn nó vào trước k_1. Đối với k_3, ta lại xét dãy chỉ gồm 2 khoá k_1, k_2 đã sắp xếp và tìm cách chèn k_3 vào dãy khoá đó để được

thứ tự sắp xếp. Một cách tổng quát, ta sẽ sắp xếp dãy k_1, k_2, …, k_i trong điều kiện dãy k_1, k_2, …, k_{i-1} đã sắp xếp rồi bằng cách chèn k_i vào dãy đó tại vị trí đúng khi sắp xếp.

```
procedure InsertionSort;
var
  i, j: Integer;
  tmp: TKey; {Biến giữ lại giá trị khoá chèn}
begin
  for i := 2 to n do {Chèn giá trị ki vào dãy k1,…, ki-1 để toàn đoạn k1, k2,…, ki trở thành đã sắp xếp}
    begin
      tmp := ki; {Giữ lại giá trị ki}
      j := i - 1;
      while (j > 0) and (tmp < kj) do {So sánh  giá trị cần chèn với lần lượt các khoá kj (i-1≥j≥0)}
        begin
          kj+1 := kj; {Đẩy lùi giá trị kj về phía sau một vị trí, tạo ra "khoảng trống" tại vị trí j}
          j := j - 1;
        end;
      kj+1 := tmp; {Đưa giá trị chèn vào "khoảng trống" mới tạo ra}
    end;
end;
```

Đối với thuật toán sắp xếp kiểu chèn, thì chi phí thời gian thực hiện thuật toán phụ thuộc vào tình trạng dãy khoá ban đầu. Nếu coi phép toán tích cực ở đây là phép so sánh $tmp < k_j$ thì:

Trường hợp tốt nhất ứng với dãy khoá đã sắp xếp rồi, mỗi lượt chỉ cần 1 phép so sánh, và như vậy tổng số phép so sánh được thực hiện là n - 1.

Trường hợp tồi tệ nhất ứng với dãy khoá đã có thứ tự ngược với thứ tự cần sắp thì ở lượt thứ i, cần có i - 1 phép so sánh và tổng số phép so sánh là:

$$(n - 1) + (n - 2) + \ldots + 1 = n * (n - 1) / 2.$$

Trường hợp các giá trị khoá xuất hiện một cách ngẫu nhiên, ta có thể coi xác suất xuất hiện mỗi khoá là đồng khả năng, thì có thể coi ở lượt thứ i, thuật toán cần trung bình i / 2 phép so sánh và tổng số phép so sánh là:

$$(1 / 2) + (2 / 2) + \ldots + (n / 2) = (n + 1) * n / 4.$$

Nhìn về kết quả đánh giá, ta có thể thấy rằng thuật toán sắp xếp kiểu chèn tỏ ra tốt hơn so với thuật toán sắp xếp chọn và sắp xếp nổi bọt. Tuy nhiên, chi phí thời gian thực hiện của thuật toán sắp xếp kiểu chèn vẫn còn khá lớn. Và xét trên phương diện tính toán lý thuyết thì **cấp của thuật toán sắp xếp kiểu chèn vẫn là O(n^2)**.

Có thể cải tiến thuật toán sắp xếp chèn nhờ nhận xét: Khi dãy khoá k_1, k_2, …, k_{i-1} đã được sắp xếp thì việc tìm vị trí chèn có thể làm bằng thuật toán tìm kiếm nhị phân và kỹ thuật chèn có thể làm bằng các lệnh dịch chuyển vùng nhớ cho nhanh. Tuy nhiên điều đó cũng không làm tốt hơn cấp độ phức tạp của thuật toán bởi trong trường hợp xấu nhất, ta phải mất n - 1 lần chèn và lần chèn thứ i ta phải dịch lùi i khoá để tạo ra khoảng trống trước khi đẩy giá trị khoá chèn vào chỗ trống đó.

```
procedure InsertionSortwithBinarySearching;
var
  i, inf, sup, median: Integer;
  tmp: TKey;
begin
  for i := 2 to n do
    begin
      tmp := ki; {Giữ lại giá trị ki}
      inf := 1; sup := i - 1; {Tìm chỗ chèn giá trị tmp vào đoạn từ kinf tới ksup+1}
      repeat {Sau mỗi vòng lặp này thì đoạn tìm bị co lại một nửa}
        median := (inf + sup) div 2; {Xét chỉ số nằm giữa chỉ số inf và chỉ số sup}
        if tmp < k[median] then sup := median - 1
        else inf := median + 1;
      until inf > sup; {Kết thúc vòng lặp thì inf = sup + 1 chính là vị trí chèn}
      <Dịch các phần tử từ kinf tới ki-1 lùi sau một vị trí>
      kinf := tmp; {Đưa giá trị tmp vào "khoảng trống" mới tạo ra}
    end;
end;
```

8.5. SHELLSORT

Nhược điểm của thuật toán sắp xếp kiểu chèn thể hiện khi mà ta luôn phải chèn một khóa vào vị trí gần đầu dãy. Trong trường hợp đó, người ta sử dụng phương pháp ShellSort.

Xét dãy khoá: $k_1, k_2, \ldots, k_n$. Với một số nguyên dương h: $1 \le h \le n$, ta có thể chia dãy đó thành h dãy con:

Dãy con 1: $k_1, k_{1+h}, k_{1+2h}, \ldots$

Dãy con 2: $k_2, k_{2+h}, k_{2+2h}, \ldots$

…

Dãy con h: $k_h, k_{2h}, k_{3h}, \ldots$

Ví dụ như dãy (4, 6, 7, 2, 3, 5, 1, 9, 8); n = 9; h = 3. Có 3 dãy con.

Dãy khoá chính:	4	6	7	2	3	5	1	9	8
Dãy con 1:	4			2			1		
Dãy con 2:		6			3			9	
Dãy con 3:			7			5			8

Những dãy con như vậy được gọi là dãy con xếp theo độ dài bước h. Tư tưởng của thuật toán ShellSort là: Với một bước h, áp dụng thuật toán sắp xếp kiểu chèn từng dãy con độc lập để làm mịn dần dãy khoá chính. Rồi lại làm tương tự đối với bước h div 2 … cho tới khi h = 1 thì ta được dãy khoá sắp xếp.

Như ở ví dụ trên, nếu dùng thuật toán sắp xếp kiểu chèn thì khi gặp khoá $k_7 = 1$, là khoá nhỏ nhất trong dãy khoá, nó phải chèn vào vị trí 1, tức là phải thao tác trên 6 khoá đứng trước nó. Nhưng nếu coi 1 là khoá của dãy con 1 thì nó chỉ cần chèn vào trước 2 khoá trong dãy con đó mà thôi. Đây chính là nguyên nhân ShellSort hiệu quả hơn sắp xếp chèn: Khoá nhỏ được nhanh chóng đưa về **gần** vị trí đúng của nó.

```
procedure ShellSort;
var
  i, j, h: Integer;
  tmp: TKey;
begin
  h := n div 2;
  while h <> 0 do {Làm mịn dãy với độ dài bước h}
    begin
      for i := h + 1 to n do
        begin {Sắp xếp chèn trên dãy con ai-h, ai, ai+h, ai+2h, …}
          tmp := ki; j := i - h;
          while (j > 0) and (kj > tmp) do
            begin
              kj+h := kj;
              j := j - h;
            end;
          kj+h := tmp;
        end;
      h := h div 2;
    end;
end;
```

8.6. THUẬT TOÁN SẮP XẾP KIỂU PHÂN ĐOẠN (QUICKSORT)

8.6.1. Tư tưởng của QuickSort

QuickSort là một phương pháp sắp xếp tốt nhất, nghĩa là dù dãy khoá thuộc kiểu dữ liệu có thứ tự nào, QuickSort cũng có thể sắp xếp được và không có một thuật toán sắp xếp nào nhanh hơn QuickSort về mặt tốc độ trung bình (theo tôi biết). Người sáng lập ra nó là C.A.R. Hoare đã mạnh dạn đặt tên cho nó là sắp xếp "NHANH".

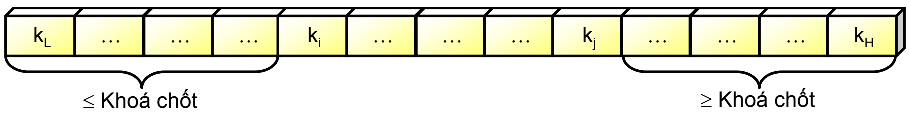
Ý tưởng chủ đạo của phương pháp có thể tóm tắt như sau: Sắp xếp dãy khoá k_1, k_2, …, k_n thì có thể coi là sắp xếp đoạn từ chỉ số 1 tới chỉ số n trong dãy khoá đó. Để sắp xếp một đoạn trong dãy khoá, nếu đoạn đó có ≤ 1 phần tử thì không cần phải làm gì cả, còn nếu đoạn đó có ít nhất 2 phần tử, ta chọn một khoá ngẫu nhiên nào đó của đoạn làm "chốt" (pivot). Mọi khoá nhỏ hơn khoá chốt được xếp vào vị trí đứng trước chốt, mọi khoá lớn hơn khoá chốt được xếp vào vị trí đứng sau chốt. Sau phép hoán chuyển như vậy thì đoạn đang xét được chia làm hai đoạn khác rỗng mà mọi khoá trong đoạn đầu đều ≤ chốt và mọi khoá trong đoạn sau đều ≥ chốt. Hay nói cách khác: Mỗi khoá trong đoạn đầu đều ≤ mọi khoá trong đoạn sau. Và vấn đề trở thành sắp xếp hai đoạn mới tạo ra (có độ dài ngắn hơn đoạn ban đầu) bằng phương pháp tương tự.

```
procedure QuickSort;

  procedure Partition(L, H: Integer); {Sắp xếp đoạn từ kL, kL+1, …, kH}
  var
    i, j: Integer;
    Pivot: TKey; {Biến lưu giá trị khoá chốt}
  begin
    if L ≥ H then Exit; {Nếu đoạn chỉ có ≤ 1 phần tử thì không phải làm gì cả}
    Pivot := kRandom(H-L+1)+L; {Chọn một khoá ngẫu nhiên trong đoạn làm khoá chốt}
    i := L; j := H; {i := vị trí đầu đoạn; j := vị trí cuối đoạn}
    repeat
      while ki < Pivot do i := i + 1; {Tìm từ đầu đoạn khoá ≥ khoá chốt}
      while kj > Pivot do j := j - 1; {Tìm từ cuối đoạn khoá ≤ khoá chốt}
      {Đến đây ta tìm được hai khoá ki và kj mà ki ≥ key ≥ kj}
      if i ≤ j then
        begin
          if i < j then {Nếu chỉ số i đứng trước chỉ số j thì đảo giá trị hai khoá ki và kj}
            <Đảo giá trị ki và kj> {Sau phép đảo này ta có: ki ≤ key ≤ kj }
          i := i + 1; j := j - 1;
        end;
    until i > j;
    Partition(L, j); Partition(i, H); {Sắp xếp hai đoạn con mới tạo ra}
  end;

begin
  Partition(1, n);
end;
```

Ta thử phân tích xem tại sao đoạn chương trình trên hoạt động đúng: Xét vòng lặp repeat…until trong lần lặp đầu tiên, **vòng lặp while thứ nhất chắc chắn sẽ tìm được khoá k_i ≥ khoá chốt** bởi chắc chắn tồn tại trong đoạn một khoá bằng khóa chốt. Tương tự như vậy, **vòng lặp while thứ hai chắc chắn tìm được khoá k_j ≤ khoá chốt.** Nếu như khoá k_i đứng trước khoá k_j thì ta đảo giá trị hai khoá, cho i tiến và j lùi. Khi đó ta có nhận xét rằng mọi khoá đứng trước vị trí i sẽ phải ≤ khoá chốt và mọi khoá đứng sau vị trí j sẽ phải ≥ khoá chốt.

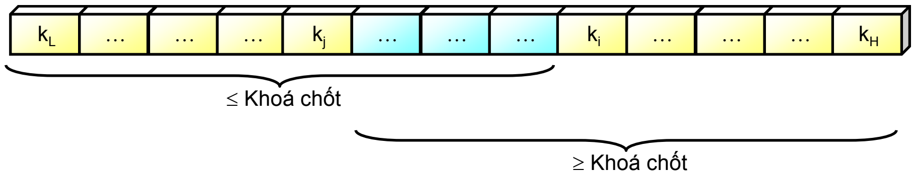


Hình 28: Vòng lặp trong của QuickSort

Điều này đảm bảo cho vòng lặp repeat…until tại bước sau, hai vòng lặp while…do bên trong chắc chắn lại tìm được hai khoá k_i và k_j mà k_i ≥ khoá chốt ≥ k_j, nếu khoá k_i đứng trước khoá k_j thì lại đảo giá trị của chúng, cho i tiến về cuối một bước và j lùi về đầu một bước. Vậy thì quá trình hoán chuyển phần tử trong vòng lặp repeat…until sẽ đảm bảo tại mỗi bước:

- Hai vòng lặp while…do bên trong luôn tìm được hai khoá k_i, k_j mà k_i ≥ khoá chốt ≥ k_j. Không có trường hợp hai chỉ số i, j chạy ra ngoài đoạn (luôn luôn có L ≤ i, j ≤ H).
- Sau mỗi phép hoán chuyển, mọi khoá đứng trước vị trí i luôn ≤ khoá chốt và mọi khoá đứng sau vị trí j luôn ≥ khoá chốt.

Vòng lặp repeat …until sẽ kết thúc khi mà chỉ số i đứng phía sau chỉ số j (Hình 29).



Hình 29: Trạng thái trước khi gọi đệ quy

Theo những nhận xét trên, nếu có một khoá nằm giữa k_j và k_i thì khoá đó phải đúng bằng khoá chốt và nó đã được đặt ở vị trí đúng của nó, nên có thể bỏ qua khoá này mà chỉ xét hai đoạn ở hai đầu. Công việc còn lại là gọi đệ quy để làm tiếp với đoạn từ k_L tới k_j và đoạn từ k_i tới k_H. Hai đoạn này ngắn hơn đoạn đang xét bởi vì $L \le j < i \le H$. Vậy thuật toán không bao giờ bị rơi vào quá trình vô hạn mà sẽ dừng và cho kết quả đúng đắn.

Xét về độ phức tạp tính toán:

Trường hợp tồi tệ nhất, là khi chọn khoá chốt, ta chọn phải khoá nhỏ nhất hay lớn nhất trong đoạn, khi đó phép phân đoạn sẽ chia thành một đoạn gồm n - 1 phần tử và đoạn còn lại chỉ có 1 phần tử. Có thể chứng minh trong trường hợp này, thời gian thực hiện giải thuật $T(n) = O(n^2)$

Trường hợp tốt nhất, phép phân đoạn tại mỗi bước sẽ chia được thành hai đoạn bằng nhau. Tức là khi chọn khoá chốt, ta chọn đúng trung vị của dãy khoá. Có thể chứng minh trong trường hợp này, thời gian thực hiện giải thuật $T(n) = O(n\log_2 n)$

Trường hợp các khoá được phân bố ngẫu nhiên, thì trung bình thời gian thực hiện giải thuật cũng là $\mathbf{T(n) = O(n\log_2 n)}$.

Việc tính toán chi tiết, đặc biệt là khi xác định T(n) trung bình, phải dùng các công cụ toán phức tạp, ta chỉ công nhận những kết quả trên.

8.6.2. Vài cải tiến của QuickSort

Việc chọn chốt cho phép phân đoạn quyết định hiệu quả của QuickSort, nếu chọn chốt không tốt, rất có thể việc phân đoạn bị suy biến thành trường hợp xấu khiến QuickSort hoạt động chậm và tràn ngăn xếp chương trình con khi gặp phải dây chuyền đệ qui quá dài. Một cải tiến sau có thể khắc phục được hiện tượng tràn ngăn xếp nhưng cũng hết sức chậm trong trường hợp xấu, kỹ thuật này khi đã phân được [L, H] được hai đoạn con [L, j] và [i, H] thì chỉ gọi đệ quy để tiếp tục đối với đoạn ngắn, và lặp lại quá trình phân đoạn đối với đoạn dài.

```
procedure QuickSort;

  procedure Partition(L, H: Integer); {Sắp xếp đoạn từ kL, kL+1, …, kH}
  var
    i, j: Integer;
  begin
    repeat
      if L ≥ H then Exit;
      <Phân đoạn [L, H] được hai đoạn con [L, j] và [i, H]>
      if <đoạn [L, j] ngắn hơn đoạn [i, H]> then
        begin
          Partition(L, j); L := i;
        end
      else
        begin
          Partition(i, H); H := j;
        end;
    until False;
  end;

begin
  Partition(1, n);
end;
```

Cải tiến thứ hai đối với QuickSort là quá trình phân đoạn nên chỉ làm đến một mức nào đó, đến khi đoạn đang xét có độ dài ≤ M (M là một số nguyên tự chọn nằm trong khoảng từ 9 tới 25) thì không phân đoạn tiếp mà nên áp dụng thuật toán sắp xếp kiểu chèn.

Cải tiến thứ ba của QuickSort là: Nên lấy trung vị của một dãy con trong đoạn để làm chốt, (trung vị của một dãy n phần tử là phần tử đứng thứ n / 2 khi sắp thứ tự). Cách chọn được đánh giá cao nhất là chọn trung vị của ba phần tử đầu, giữa và cuối đoạn.

Cuối cùng, ta có nhận xét: QuickSort là một công cụ sắp xếp mạnh, chỉ có điều khó chịu gặp phải là trường hợp suy biến của QuickSort (quá trình phân đoạn chia thành một dãy rất ngắn và một dãy rất dài). Và điều này trên phương diện lý thuyết là không thể khắc phục được: Ví dụ với n = 10000.

Nếu như chọn chốt là khoá đầu đoạn (Thay dòng chọn khoá chốt bằng Pivot := k_L) hay chọn chốt là khoá cuối đoạn (Thay bằng Pivot := k_H) thì với dãy sau, chương trình hoạt động rất chậm:

(1, 2, 3, 4, 5, …, 9999, 10000)

Nếu như chọn chốt là khoá giữa đoạn (Thay dòng chọn khoá chốt bằng Pivot := $k_{(L+H) \text{ div } 2}$) thì với dãy sau, chương trình cũng rất chậm:

(1, 2, …, 4999, 5000, 5000, 4999, …, 2, 1)

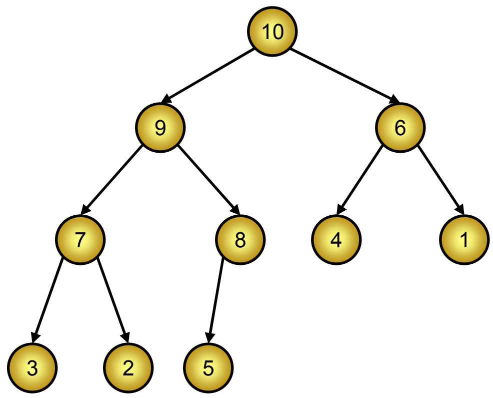
Trong trường hợp chọn chốt là trung vị dãy con hay chọn chốt ngẫu nhiên, thật khó có thể tìm ra một bộ dữ liệu khiến cho QuickSort hoạt động chậm. Nhưng ta cũng cần hiểu rằng với mọi chiến lược chọn chốt, trong 10000! dãy hoán vị của dãy (1, 2, … 10000) thế nào cũng có một dãy làm QuickSort bị suy biến, tuy nhiên trong trường hợp chọn chốt ngẫu nhiên, xác suất xảy ra dãy này quá nhỏ tới mức ta không cần phải tính đến, như vậy khi đã chọn chốt ngẫu nhiên

thì ta không cần phải quan tâm tới ngăn xếp đệ quy, không cần quan tâm tới kỹ thuật khử đệ quy và vấn đề suy biến của QuickSort.

8.7. THUẬT TOÁN SẮP XẾP KIỂU VUN ĐỐNG (HEAPSORT)

8.7.1. Đống (heap)

Đống là một dạng cây nhị phân hoàn chỉnh đặc biệt mà giá trị lưu tại mọi nút nhánh đều lớn hơn hay bằng giá trị lưu trong hai nút con của nó.



Hình 30: Heap

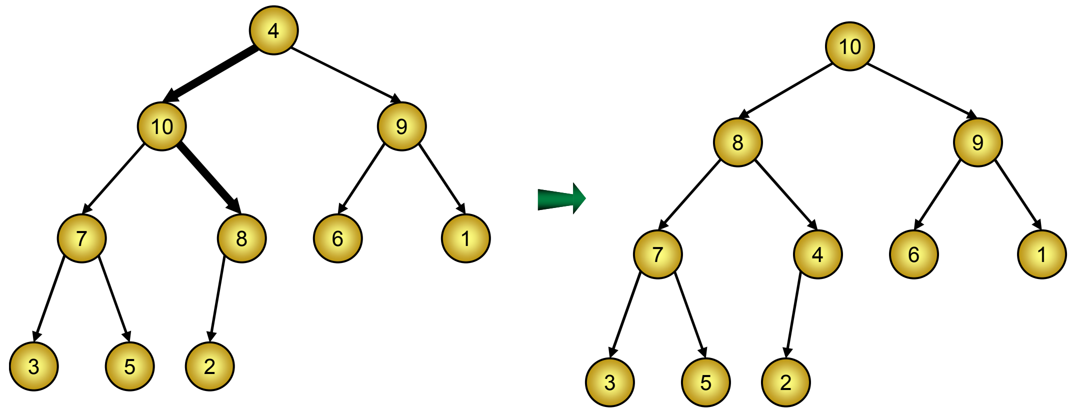
8.7.2. Vun đống

Trong bài học về cây, ta đã biết một dãy khoá $k_1, k_2, \ldots, k_n$ là biểu diễn của một cây nhị phân hoàn chỉnh mà k_i là giá trị lưu trong nút thứ i, nút con của nút thứ i là nút 2i và nút 2i + 1, nút cha của nút thứ j là nút j div 2. Vấn đề đặt ra là sắp lại dãy khoá đã cho để nó biểu diễn một đống.

Vì cây nhị phân chỉ gồm có một nút hiển nhiên là đống, nên **để vun một nhánh cây gốc r thành đống, ta có thể coi hai nhánh con của nó (nhánh gốc 2r và 2r + 1) đã là đống rồi** và thực hiện thuật toán vun đống từ dưới lên (bottom-up) đối với cây: Gọi h là chiều cao của cây, nút ở mức h (nút lá) đã là gốc một đống, ta vun lên để những nút ở mức h - 1 cũng là gốc của đống, … cứ như vậy cho tới nút ở mức 1 (nút gốc) cũng là gốc của đống.

Thuật toán vun thành đống đối với cây gốc r, hai nhánh con của r đã là đống rồi:

Giả sử ở nút r chứa giá trị V. Từ r, ta cứ đi tới nút con chứa giá trị lớn nhất trong 2 nút con, cho tới khi gặp phải một nút c mà mọi nút con của c đều chứa giá trị ≤ V (nút lá cũng là trường hợp riêng của điều kiện này). Dọc trên đường đi từ r tới c, ta đẩy giá trị chứa ở nút con lên nút cha và đặt giá trị V vào nút c.

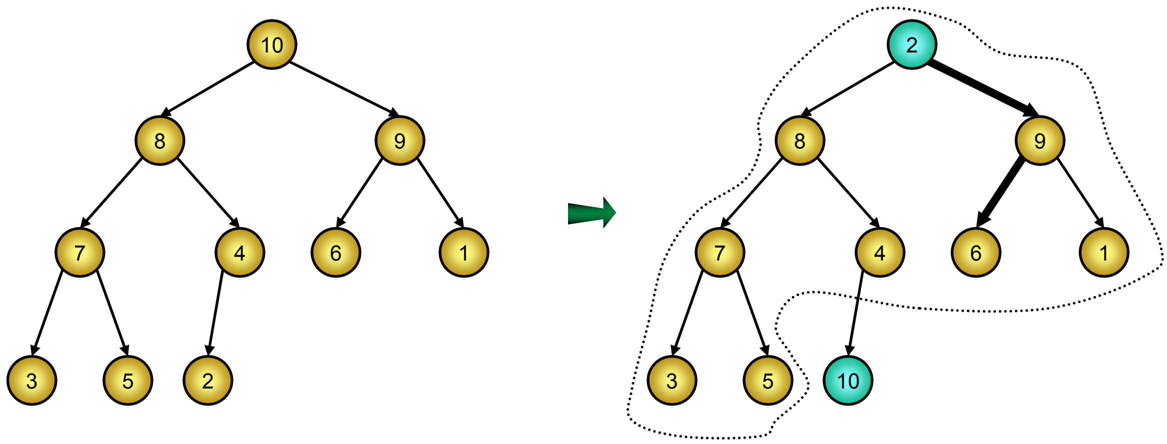


Hình 31: Vun đống

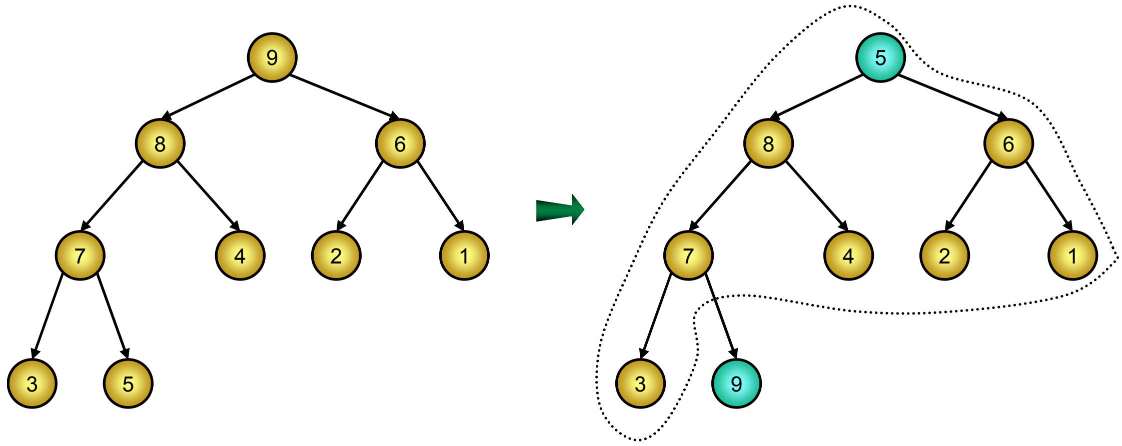
8.7.3. Tư tưởng của HeapSort

Đầu tiên, dãy khoá k_1, k_2, …, k_n được vun từ dưới lên để nó biểu diễn một đống, khi đó khoá k_1 tương ứng với nút gốc của đống là khoá lớn nhất, ta đảo giá trị khoá đó cho k_n và không tính tới k_n nữa (Hình 32). Còn lại dãy khoá k_1, k_2, …, k_{n-1} tuy không còn là biểu diễn của một đống nữa nhưng nó lại biểu diễn cây nhị phân hoàn chỉnh mà hai nhánh cây ở nút thứ 2 và nút thứ 3 (hai nút con của nút 1) đã là đống rồi. Vậy chỉ cần vun một lần, ta lại được một đống, đảo giá trị k_1 cho k_{n-1} và tiếp tục cho tới khi đống chỉ còn lại 1 nút (Hình 33).

Ví dụ:



Hình 32: Đảo giá trị k_1 cho k_n và xét phần còn lại



Hình 33: Vun phần còn lại thành đống rồi lại đảo trị k_1 cho k_{n-1}

Thuật toán HeapSort có hai thủ tục chính:

Thủ tục Adjust(root, endnode) vun cây gốc root thành đống trong điều kiện hai cây gốc 2.root và 2.root +1 đã là đống rồi. Các nút từ endnode + 1 tới n đã nằm ở vị trí đúng và không được tính tới nữa.

Thủ tục HeapSort mô tả lại quá trình vun đống và chọn phần tử theo ý tưởng trên:

```
procedure HeapSort;
var
  r, i: Integer;

  procedure Adjust(root, endnode: Integer); {Vun cây gốc Root thành đống}
  var
    c: Integer;
    Key: TKey; {Biến lưu giá trị khoá ở nút Root}
  begin
    Key := k_root;
    while root * 2 ≤ endnode do {Chừng nào root chưa phải là lá}
      begin
        c := Root * 2; {Xét nút con trái của Root, so sánh với giá trị nút con phải, chọn ra nút mang giá trị lớn nhất}
        if (c < endnode) and (k_c < k_c+1) then c := c + 1;
        if k_c ≤ Key then Break; {Cả hai nút con của Root đều mang giá trị ≤ Key thì dừng ngay}
        k_root := k_c; root := c; {Chuyển giá trị từ nút con c lên nút cha root và đi xuống xét nút con c}
      end;
    k_root := Key; {Đặt giá trị Key vào nút root}
  end;

begin {Bắt đầu thuật toán HeapSort}
  for r := n div 2 downto 1 do Adjust(r, n); {Vun cây từ dưới lên tạo thành đống}
  for i := n downto 2 do
    begin
      <Đảo giá trị k_1 và k_i> {Khoá lớn nhất được chuyển ra cuối dãy}
      Adjust(1, i - 1); {Vun phần còn lại thành đống}
    end;
end;
```

Về độ phức tạp của thuật toán, ta đã biết rằng cây nhị phân hoàn chỉnh có n nút thì chiều cao của nó không quá [log2(n + 1)] + 1. Cứ cho là trong trường hợp xấu nhất thủ tục Adjust phải thực hiện tìm đường đi từ nút gốc tới nút lá ở xa nhất thì đường đi tìm được cũng chỉ dài bằng chiều cao của cây và độ phức tạp của một lần gọi Adjust là O(log2n). Từ đó có thể suy ra, trong trường hợp xấu nhất, **độ phức tạp của HeapSort cũng chỉ là O(n$\log_2$n)**. Việc đánh giá

thời gian thực hiện trung bình phức tạp hơn, ta chỉ ghi nhận một kết quả đã chứng minh được là độ phức tạp trung bình của HeapSort cũng là $O(n\log_2 n)$.

Có thể nhận xét thêm là QuickSort đệ quy cần thêm không gian nhớ cho Stack, còn HeapSort ngoài một nút nhớ phụ để thực hiện việc đổi chỗ, nó không cần dùng thêm gì khác. HeapSort tốt hơn QuickSort về phương diện lý thuyết bởi không có trường hợp tồi tệ nào HeapSort có thể mắc phải. Cũng nhờ có HeapSort mà giờ đây khi giải mọi bài toán có chứa mô-đun sắp xếp, ta có thể nói rằng độ phức tạp của thủ tục sắp xếp đó không quá $O(n\log_2 n)$.

8.8. SẮP XẾP BẰNG PHÉP ĐẾM PHÂN PHỐI (DISTRIBUTION COUNTING)

Có một thuật toán sắp xếp đơn giản cho trường hợp đặc biệt: Dãy khoá $k_1, k_2, …, k_n$ là các số nguyên nằm trong khoảng từ 0 tới M (TKey = 0..M).

Ta dựng dãy $c_0, c_1, …, c_M$ các biến đếm, ở đây c_V là số lần xuất hiện giá trị V trong dãy khoá:

```
for V := 0 to M do c_V := 0; {Khởi tạo dãy biến đếm}
for i := 1 to n do c_k_i := c_k_i + 1;
```

Ví dụ với dãy khoá: 1, 2, 2, 3, 0, 0, 1, 1, 3, 3 (n = 10, M = 3), sau bước đếm ta có:

$c_0 = 2; c_1 = 3; c_2 = 2; c_3 = 3$.

Dựa vào dãy biến đếm, ta hoàn toàn có thể biết được: sau khi sắp xếp thì giá trị V phải nằm từ vị trí nào tới vị trí nào. Như ví dụ trên thì giá trị 0 phải nằm từ vị trí 1 tới vị trí 2; giá trị 1 phải đứng liên tiếp từ vị trí 3 tới vị trí 5; giá trị 2 đứng ở vị trí 6 và 7 còn giá trị 3 nằm ở ba vị trí cuối 8, 9, 10:

0 0 1 1 1 2 2 3 3 3

Tức là sau khi sắp xếp:

Giá trị 0 đứng trong đoạn từ vị trí 1 tới vị trí c_0.

Giá trị 1 đứng trong đoạn từ vị trí $c_0 + 1$ tới vị trí $c_0 + c_1$.

Giá trị 2 đứng trong đoạn từ vị trí $c_0 + c_1 + 1$ tới vị trí $c_0 + c_1 + c_2$.

…

Giá trị v trong đoạn đứng từ vị trí $c_0 + c_1 + … + c_{v-1} + 1$ tới vị trí $c_0 + c_1 + c_2 + … + c_v$.

…

Để ý vị trí cuối của mỗi đoạn, nếu ta tính lại dãy c như sau:

```
for V := 1 to M do c_V := c_V-1 + c_V
```

Thì **c_V là vị trí cuối của đoạn chứa giá trị V trong dãy khoá đã sắp xếp.**

Muốn dựng lại dãy khoá sắp xếp, ta thêm một dãy khoá phụ x1, x2, …, xn. Sau đó duyệt lại dãy khoá k, mỗi khi gặp khoá mang giá trị V ta đưa giá trị đó vào khoá x_{c_v} và giảm cv đi 1.

```
for i := n downto 1 do
  begin
    V := k_i;
    X_c_V := k_i; c_V := c_V - 1;
  end;
```

Khi đó dãy khoá x chính là dãy khoá đã được sắp xếp, công việc cuối cùng là gán giá trị dãy khoá x cho dãy khoá k.

```
procedure DistributionCounting; {TKey = 0..M}
var
  c: array[0..M] of Integer; {Dãy biến đếm số lần xuất hiện mỗi giá trị}
  x: TArray; {Dãy khoá phụ}
  i: Integer;
  V: TKey;
begin
  for V := 0 to M do c_V := 0; {Khởi tạo dãy biến đếm}
  for i := 1 to n do c_{k_i} := c_{k_i} + 1; {Đếm số lần xuất hiện các giá trị}
  for V := 1 to M do c_V := c_{V-1} + c_V; {Tính vị trí cuối mỗi đoạn}
  for i := n downto 1 do
    begin
      V := k_i;
      x_{c_V} := k_i; c_V := c_V - 1;
    end;
  k := x; {Sao chép giá trị từ dãy khoá x sang dãy khoá k}
end;
```

Rõ ràng độ phức tạp của phép đếm phân phối là O(max(M, n)). Nhược điểm của phép đếm phân phối là khi M quá lớn thì cho dù n nhỏ cũng không thể làm được.

Có thể có thắc mắc tại sao trong thao tác dựng dãy khoá x, phép duyệt dãy khoá k theo thứ tự nào thì kết quả sắp xếp cũng như vậy, vậy tại sao ta lại chọn phép duyệt ngược từ dưới lên?. Để trả lời câu hỏi này, ta phải phân tích thêm một đặc trưng của các thuật toán sắp xếp:

8.9. TÍNH ỔN ĐỊNH CỦA THUẬT TOÁN SẮP XẾP (STABILITY)

Một phương pháp sắp xếp được gọi là **ổn định** nếu nó bảo toàn thứ tự ban đầu của các bản ghi mang khoá bằng nhau trong danh sách. Ví dụ như ban đầu danh sách sinh viên được xếp theo thứ tự tên alphabet, thì khi sắp xếp danh sách sinh viên theo thứ tự giảm dần của điểm thi, những sinh viên bằng điểm nhau sẽ được dồn về một đoạn trong danh sách và vẫn được giữ nguyên thứ tự tên alphabet.

Hãy xem lại nhưng thuật toán sắp xếp ở trước, trong những thuật toán đó, thuật toán sắp xếp nổi bọt, thuật toán sắp xếp chèn và phép đếm phân phối là những thuật toán sắp xếp ổn định, còn những thuật toán sắp xếp khác (và nói chung những thuật toán sắp xếp đòi hỏi phải đảo giá trị 2 bản ghi ở vị trí bất kỳ) là không ổn định.

Với phép đếm phân phối ở mục trước, ta nhận xét rằng nếu hai bản ghi có khoá sắp xếp bằng nhau thì khi đưa giá trị vào dãy bản ghi phụ, bản ghi nào vào trước sẽ nằm phía sau. Vậy nên ta sẽ đẩy giá trị các bản ghi vào dãy phụ theo thứ tự ngược để giữ được thứ tự tương đối ban đầu.

Nói chung, mọi phương pháp sắp xếp tổng quát cho dù không ổn định thì đều có thể biến đổi để nó trở thành ổn định, phương pháp chung nhất được thể hiện qua ví dụ sau:

Giả sử ta cần sắp xếp các sinh viên trong danh sách theo thứ tự giảm dần của điểm bằng một thuật toán sắp xếp ổn định. Ta thêm cho mỗi sinh viên một khoá Index là thứ tự ban đầu của

anh ta trong danh sách. Trong thuật toán sắp xếp được áp dụng, cứ chỗ nào cần so sánh hai sinh viên A và B xem anh nào phải đứng trước, trước hết ta quan tâm tới điểm số: Nếu điểm của A khác điểm của B thì anh nào điểm cao hơn sẽ đứng trước, nếu điểm số bằng nhau thì anh nào có Index nhỏ hơn sẽ đứng trước.

Trong một số bài toán, tính ổn định của thuật toán sắp xếp quyết định tới cả tính đúng đắn của toàn thuật toán lớn. Chính tính "nhanh" của QuickSort và tính ổn định của phép đếm phân phối là cơ sở nền tảng cho hai thuật toán sắp xếp cực nhanh trên các dãy khoá số mà ta sẽ trình bày dưới đây.

8.10. THUẬT TOÁN SẮP XẾP BẰNG CƠ SỐ (RADIXSORT)

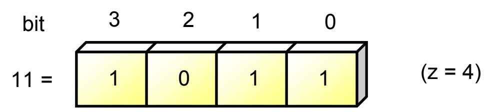
Bài toán đặt ra là: Cho dãy khoá là các số tự nhiên $k_1, k_2, \ldots, k_n$ hãy sắp xếp chúng theo thứ tự không giảm. (Trong trường hợp ta đang xét, TKey là kiểu số tự nhiên)

8.10.1. Sắp xếp cơ số theo kiểu hoán vị các khoá (Exchange RadixSort)

Hãy xem lại thuật toán QuickSort, tại bước phân đoạn nó phân đoạn đang xét thành hai đoạn thoả mãn mỗi khoá trong đoạn đầu ≤ mọi khoá trong đoạn sau và thực hiện tương tự trên hai đoạn mới tạo ra, việc phân đoạn được tiến hành với sự so sánh các khoá với giá trị một khoá chốt.

Đối với các số nguyên thì ta có thể coi mỗi số nguyên là một dãy z bit đánh số từ bit 0 (bit ở hàng đơn vị) tới bit z - 1 (bit cao nhất).

Ví dụ:



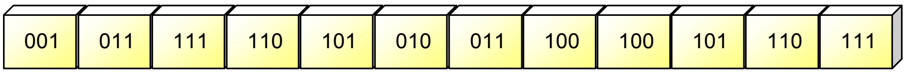
Hình 34: Đánh số các bit

Vậy thì tại bước phân đoạn dãy khoá từ k_1 tới k_n, ta có thể đưa những khoá có bit cao nhất là 0 về đầu dãy, những khoá có bit cao nhất là 1 về cuối dãy. Dễ thấy rằng những khoá bắt đầu bằng bit 0 sẽ phải nhỏ hơn những khoá bắt đầu bằng bit 1. Tiếp tục quá trình phân đoạn với hai đoạn dãy khoá: Đoạn gồm các khoá có bit cao nhất là 0 và đoạn gồm các khoá có bit cao nhất là 1. Với những khoá thuộc cùng một đoạn thì có bit cao nhất giống nhau, nên ta có thể áp dụng quá trình phân đoạn tương tự trên theo bit thứ z - 2 và cứ tiếp tục như vậy …

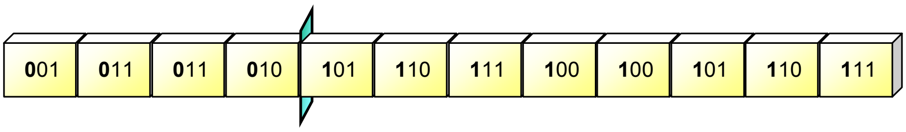
Quá trình phân đoạn kết thúc nếu như đoạn đang xét là rỗng hay ta đã tiến hành phân đoạn đến tận bit đơn vị, tức là tất cả các khoá thuộc một trong hai đoạn mới tạo ra đều có bit đơn vị bằng nhau (điều này đồng nghĩa với sự bằng nhau ở tất cả những bit khác, tức là bằng nhau về giá trị khoá).

Ví dụ:

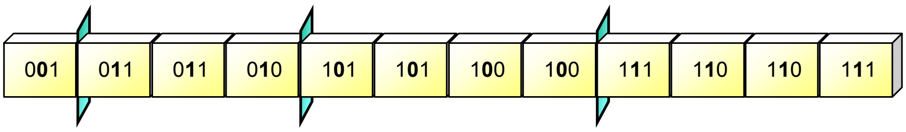
Xét dãy khoá: 1, 3, 7, 6, 5, 2, 3, 4, 4, 5, 6, 7. Tương ứng với các dãy 3 bit:



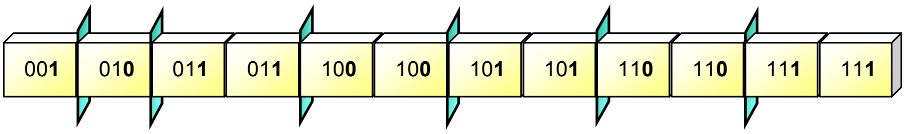
Trước hết ta chia đoạn dựa vào bit 2 (bit cao nhất):



Sau đó chia tiếp hai đoạn tạo ra dựa vào bit 1:



Cuối cùng, chia tiếp những đoạn tạo ra dựa vào bit 0:



Ta được dãy khoá tương ứng: 1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7 là dãy khoá sắp xếp.

Quá trình chia đoạn dựa vào bit b có thể chia thành một đoạn rỗng và một đoạn gồm toàn bộ các phần tử còn lại, nhưng việc chia đoạn không bao giờ bị rơi vào quá trình đệ quy vô hạn bởi những lần đệ quy tiếp theo sẽ phân đoạn dựa vào bit b - 1, b - 2 …và nếu xét đến bit 0 sẽ phải dừng lại. Công việc còn lại là cố gắng hiểu đoạn chương trình sau và phân tích xem tại sao nó hoạt động đúng:

```
procedure ExchangeRadixSort;
var
  z: Integer; {Độ dài dãy bit biểu diễn mỗi khoá}

  procedure Partition(L, H, b: Integer); {Phân đoạn [L, H] dựa vào bit b}
  var
    i, j: Integer;
  begin
    if L ≥ H then Exit;
    i := L; j := H;
    repeat
      {Hai vòng lặp trong dưới đây luôn cầm canh i < j}
      while (i < j) and (Bit b của k_i = 0) do i := i + 1; {Tìm khoá có bit b = 1 từ đầu đoạn}
      while (i < j) and (Bit b của k_j = 1) do j := j - 1; {Tìm khoá có bit b = 0 từ cuối đoạn}
      <Đảo giá trị k_i cho k_j>;
    until i = j;
    if <Bit b của k_j = 0> then j := j + 1; {j là điểm bắt đầu của đoạn có bit b là 1}
    if b > 0 then {Chưa xét tới bit đơn vị}
      begin
        Partition(L, j - 1, b - 1); Partition(j, R, b - 1);
      end;
  end;

begin
  <Dựa vào giá trị lớn nhất của dãy khoá,
   xác định z là độ dài dãy bit biểu diễn mỗi khoá>
  Partition(1, n, z - 1);
end;
```

Với RadixSort, ta hoàn toàn có thể làm trên hệ cơ số R khác chứ không nhất thiết phải làm trên hệ nhị phân (ý tưởng cũng tương tự như trên), tuy nhiên quá trình phân đoạn sẽ không phải chia làm 2 mà chia thành R đoạn. Về độ phức tạp của thuật toán, ta thấy để phân đoạn bằng một bit thì thời gian sẽ là C.n để chia tất cả các đoạn cần chia bằng bit đó (C là hằng số). Vậy tổng thời gian phân đoạn bằng z bit sẽ là C.n.z. **Trong trường hợp xấu nhất, độ phức tạp của RadixSort là O(n.z)**. Và **độ phức tạp trung bình của RadixSort là O(n.min(z, $\log_2$n))**.

Nói chung, RadixSort cài đặt như trên chỉ thể hiện tốc độ tối đa trên các hệ thống cho phép xử lý trực tiếp trên các bit: Hệ thống phải cho phép lấy một bit ra dễ dàng và thao tác với thời gian nhanh hơn hẳn so với thao tác trên Byte và Word. Khi đó RadixSort sẽ tốt hơn nhiều QuickSort. (Ta thử lập trình sắp xếp các dãy nhị phân độ dài z theo thứ tự từ điển để khảo sát). Trên các máy tính hiện nay chỉ cho phép xử lý trực tiếp trên Byte (hay Word, DWord v.v…), việc tách một bit ra khỏi Byte đó để xử lý lại rất chậm và làm ảnh hưởng không nhỏ tới tốc độ của RadixSort. Chính vì vậy, tuy đây là một phương pháp hay, nhưng khi cài đặt cụ thể thì tốc độ cũng chỉ ngang ngửa chứ không thể qua mặt QuickSort được.

8.10.2. Sắp xếp cơ số trực tiếp (Straight RadixSort)

Ta sẽ trình bày phương pháp sắp xếp cơ số trực tiếp bằng một ví dụ: Sắp xếp dãy khoá:

925	817	821	638	639	744	742	563	570	166

Trước hết, ta sắp xếp dãy khoá này theo thứ tự tăng dần của chữ số hàng đơn vị bằng một thuật toán sắp xếp khác, được dãy khoá:

570	821	742	563	744	925	166	817	638	639

Sau đó, ta sắp xếp dãy khoá mới tạo thành theo thứ tự tăng dần của chữ số hàng chục bằng một thuật toán sắp xếp **ổn định**, được dãy khoá:

817	821	925	638	639	742	744	563	166	570

Vì thuật toán sắp xếp ta sử dụng là ổn định, nên nếu hai khoá có chữ số hàng chục giống nhau thì khoá nào có chữ số hàng đơn vị nhỏ hơn sẽ đứng trước. Nói như vậy có nghĩa là dãy khoá thu được sẽ có thứ tự tăng dần về giá trị tạo thành từ hai chữ số cuối.

Cuối cùng, ta sắp xếp lại dãy khoá theo thứ tự tăng dần của chữ số hàng trăm cũng bằng một thuật toán sắp xếp ổn định, thu được dãy khoá:

166	563	570	638	639	742	744	817	821	925

Lập luận tương tự như trên dựa vào tính ổn định của phép sắp xếp, dãy khoá thu được sẽ có thứ tự tăng dần về giá trị tạo thành bởi cả ba chữ số, đó là dãy khoá đã sắp.

Nhận xét:

Ta hoàn toàn có thể coi số chữ số của mỗi khoá là bằng nhau, như ví dụ trên nếu có số 15 trong dãy khoá thì ta có thể coi nó là 015.

Cũng từ ví dụ, ta có thể thấy rằng số lượt thao tác sắp xếp phải áp dụng đúng bằng số chữ số tạo thành một khoá. Với một hệ cơ số lớn, biểu diễn một giá trị khoá sẽ phải dùng ít chữ số hơn. Ví dụ số 12345 trong hệ thập phân phải dùng tới 5 chữ số, còn trong hệ cơ số 1000 chỉ cần dùng 2 chữ số AB mà thôi, ở đây A là chữ số mang giá trị 12 còn B là chữ số mang giá trị 345.

Tốc độ của sắp xếp cơ số trực tiếp phụ thuộc rất nhiều vào thuật toán sắp xếp ổn định tại mỗi bước. Không có một lựa chọn nào khác tốt hơn phép đếm phân phối. Tuy nhiên, phép đếm phân phối có thể không cài đặt được hoặc kém hiệu quả nếu như tập giá trị khoá quá rộng, không cho phép dựng ra dãy các biến đếm hoặc phải sử dụng dãy biến đếm quá dài (Điều này xảy ra nếu chọn hệ cơ số quá lớn).

Một lựa chọn khôn ngoan là nên chọn hệ cơ số thích hợp cho từng trường hợp cụ thể để dung hoà tới mức tối ưu nhất ba mục tiêu:

Việc lấy ra một chữ số của một số được thực hiện dễ dàng

Sử dụng ít lần gọi phép đếm phân phối.

Phép đếm phân phối thực hiện nhanh

```
procedure StraightRadixSort;
const
  radix = …; {Tuỳ chọn hệ cơ số radix cho hợp lý}
var
  t: TArray; {Dãy khoá phụ}
  p: Integer;
  nDigit: Integer; {Số chữ số cho một khoá, đánh số từ chữ số thứ 0 là hàng đơn vị đến chữ số thứ nDigit - 1}
  Flag: Boolean; {Flag = True thì sắp dãy  k, ghi kết quả vào dãy t; Flag = False thì sắp dãy t, ghi kq vào k}

  function GetDigit(Num: TKey; p: Integer): Integer; {Lấy chữ số thứ p của số Num (0≤p<nDigit)}
  begin
    GetDigit := Num div radix^p mod radix; {Trường hợp cụ thể có thể có cách viết tốt hơn}
  end;

  {Sắp xếp ổn định dãy số x theo thứ tự tăng dần của chữ số thứ p, kết quả sắp xếp được chứa vào dãy số y}
  procedure DCount(var x, y: TArray; p: Integer); {Thuật toán đếm phân phối, sắp từ x sang y}
  var
    c: array[0..radix - 1] of Integer; {c_d là số lần xuất hiện chữ số d tại vị trí p}
    i, d: Integer;
  begin
    for d := 0 to radix - 1 do c_d := 0;
    for i := 1 to n do
      begin
        d := GetDigit(x_i, p); c_d := c_d + 1;
      end;
    for d := 1 to radix - 1 do c_d := c_(d-1) + c_d; {các c_d trở thành các mốc cuối đoạn}
    for i := n downto 1 do {Điền giá trị vào dãy y}
      begin
        d := GetDigit(x_i, p);
        y_(c_d) := x_i; c_d := c_d - 1;
      end;
  end;

begin {Thuật toán sắp xếp cơ số trực tiếp}
  <Dựa vào giá trị lớn nhất trong dãy khoá,
   xác định nDigit là số chữ số phải dùng cho mỗi khoá trong hệ radix>;
  Flag := True;
  for p := 0 to nDigit - 1 do {Xét từ chữ số hàng đơn vị lên, sắp xếp ổn định theo chữ số thứ p}
    begin
      if Flag then DCount(k, t, p) else DCount(t, k, p);
      Flag := not Flag; {Đảo cờ, dùng k tính t rồi lại dùng t tính k …}
    end;
  if not Flag then k := t; {Nếu kết quả cuối cùng đang ở trong t thì sao chép giá trị từ t sang k}
end;
```

Xét phép đếm phân phối, ta đã biết độ phức tạp của nó là O(max(radix, n)). Mà radix là một hằng số tự ta chọn từ trước, nên khi n lớn, độ phức tạp của phép đếm phân phối là O(n). Thuật toán sử dụng nDigit lần phép đếm phân phối nên có thể thấy **độ phức tạp của thuật toán là O(n.nDigit)** bất kể dữ liệu đầu vào.

Ta có thể coi sắp xếp cơ số trực tiếp là một mở rộng của phép đếm phân phối, khi dãy số chỉ toàn các số có 1 chữ số (trong hệ radix) thì đó chính là phép đếm phân phối. Sự khác biệt ở đây là: Sắp xếp cơ số trực tiếp có thể thực hiện với các khoá mang giá trị lớn; còn phép đếm phân phối chỉ có thể làm trong trường hợp các khoá mang giá trị nhỏ, bởi nó cần một lượng bộ nhớ đủ rộng để giăng ra dãy biến đếm số lần xuất hiện cho từng giá trị.

8.11. THUẬT TOÁN SẮP XẾP TRỘN (MERGESORT)

8.11.1. Phép trộn 2 đường

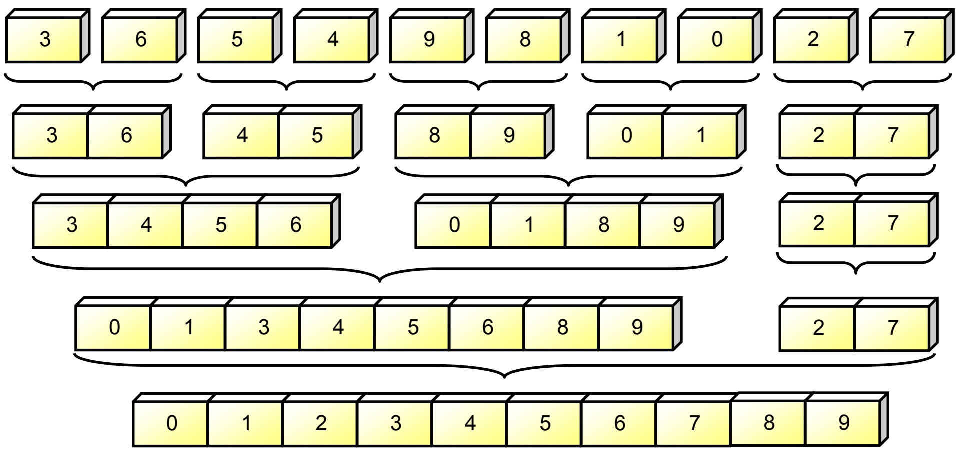
Phép trộn 2 đường là phép hợp nhất hai dãy khoá **đã sắp xếp** để ghép lại thành một dãy khoá có kích thước bằng tổng kích thước của hai dãy khoá ban đầu và dãy khoá tạo thành cũng có thứ tự sắp xếp. Nguyên tắc thực hiện của nó khá đơn giản: so sánh hai khoá đứng đầu hai dãy, chọn ra khoá nhỏ nhất và đưa nó vào miền sắp xếp (một dãy khoá phụ có kích thước bằng tổng kích thước hai dãy khoá ban đầu) ở vị trí thích hợp. Sau đó, khoá này bị loại ra khỏi dãy khoá chứa nó. Quá trình tiếp tục cho tới khi một trong hai dãy khoá đã cạn, khi đó chỉ cần chuyển toàn bộ dãy khoá còn lại ra miền sắp xếp là xong.

Ví dụ: Với hai dãy khoá: (1, 3, 10, 11) và (2, 4, 9)

Dãy 1	Dãy 2	Khoá nhỏ nhất trong 2 dãy	Miền sắp xếp
(1, 3, 10, 11)	(2, 4, 9)	1	(1)
(3, 10, 11)	(2, 4, 9)	2	(1, 2)
(3, 10, 11)	(4, 9)	3	(1, 2, 3)
(10, 11)	(4, 9)	4	(1, 2, 3, 4)
(10, 11)	(9)	9	(1, 2, 3, 4, 9)
(10, 11)	Ø	Dãy 2 là Ø, đưa nốt dãy 1 vào miền sắp xếp	(1, 2, 3, 4, 9, 10, 11)

8.11.2. Sắp xếp bằng trộn 2 đường trực tiếp

Ta có thể coi mỗi khoá trong dãy khoá $k_1, k_2, \ldots, k_n$ là một mạch với độ dài 1, dĩ nhiên các mạch độ dài 1 có thể coi là đã được sắp. Nếu trộn hai mạch liên tiếp lại thành một mạch có độ dài 2, ta lại được dãy gồm các mạch đã được sắp. Cứ tiếp tục như vậy, số mạch trong dãy sẽ giảm dần sau mỗi lần trộn (Hình 35)



Hình 35: Thuật toán sắp xếp trộn

Để tiến hành thuật toán sắp xếp trộn hai đường trực tiếp, ta viết các thủ tục:

Thủ tục Merge(var x, y: TArray; a, b, c: Integer); thủ tục này trộn mạch $x_a, x_{a+1}, \ldots, x_b$ với mạch $x_{b+1}, x_{b+2} \ldots, x_c$ để được mạch $y_a, y_{a+1}, \ldots, y_c$.

Thủ tục MergeByLength(var x, y: TArray; len: Integer); thủ tục này trộn lần lượt các cặp mạch theo thứ tự:

Trộn mạch $x_1 \ldots x_{len}$ và $x_{len+1} \ldots x_{2len}$ thành mạch $y_1 \ldots y_{2len}$.

Trộn mạch $x_{2len+1} \ldots x_{3len}$ và $x_{3len+1} \ldots x_{4len}$ thành mạch $y_{2len+1} \ldots y_{4len}$.

…

Lưu ý rằng đến cuối cùng ta có thể gặp hai trường hợp: Hoặc còn lại hai mạch mà mạch thứ hai có độ dài < len. Hoặc chỉ còn lại một mạch. Trường hợp thứ nhất ta phải quản lý chính xác các chỉ số để thực hiện phép trộn, còn trường hợp thứ hai thì không được quên thao tác đưa thẳng mạch duy nhất còn lại sang dãy y.

Cuối cùng là thủ tục MergeSort, thủ tục này cần một dãy khoá phụ $t_1, t_2, \ldots, t_n$. Trước hết ta gọi MergeByLength(k, t, 1) để trộn hai phần tử liên tiếp của k thành một mạch trong t, sau đó lại gọi MergeByLength(t, k, 2) để trộn hai mạch liên tiếp trong t thành một mạch trong k, rồi lại gọi MergeByLength(k, t, 4) để trộn hai mạch liên tiếp trong k thành một mạch trong t …Như vậy k và t được sử dụng với vai trò luân phiên: một dãy chứa các mạch và một dãy dùng để trộn các cặp mạch liên tiếp để được mạch lớn hơn.

```
procedure MergeSort;
var
  t: TArray; {Dãy khoá phụ}
  len: Integer;
  Flag: Boolean; {Flag = True: trộn các mạch trong k vào t; Flag = False: trộn các mạch trong t vào k}

  procedure Merge(var X, Y: TArray; a, b, c: Integer);{Trộn Xa…Xb và Xb+1…Xc}
  var
    i, j, p: Integer;
  begin
    {Chỉ số p chạy trong miền sắp xếp, i chạy theo mạch thứ nhất, j chạy theo mạch thứ hai}
    p := a; i := a; j := b + 1;
    while (i ≤ b) and (j ≤ c) then {Chừng nào cả hai mạch đều chưa xét hết}
      begin
        if Xi ≤ Xj then {So sánh hai phần tử nhỏ nhất trong hai mạch mà chưa bị đưa vào miền sắp xếp}
          begin
            Yp := Xi; i := i + 1; {Đưa xi vào miền sắp xếp và cho i chạy}
          end
        else
          begin
            Yp := Xj; j := j + 1; {Đưa xj vào miền sắp xếp và cho j chạy}
          end;
        p := p + 1;
      end;
    if i ≤ b then (Yp, Yp+1, …, Yc) := (Xi, Xi+1, …, Xb) {Mạch 2 hết trước, Đưa phần cuối của mạch 1 vào miền sắp xếp}
    else (Yp, Yp+1, …, Yc) := (Xj, Xj+1, …, Xc); {Mạch 1 hết trước, Đưa phần cuối của mạch 2 vào miền sắp xếp}
  end;

  procedure MergeByLength(var X, Y: TArray; len: Integer);
  begin
    a := 1; b := len; c := 2 * len;
    while c ≤ n do {Trộn hai mạch xa…xb và xb+1…xc đều có độ dài len}
      begin
        Merge(X, Y, a, b, c);
        a := a + 2 * len; b := b + 2 * len; c := c + 2 * len; {Dịch các chỉ số a, b, c về sau 2.len vị trí}
      end;
    if b < n then Merge(X, Y, a, b, n) {Còn lại hai mạch mà mạch thứ hai có độ dài ngắn hơn len}
    else
      if a ≤ n then (Ya, Ya+1, …, Yn) := (Xa, Xa+1, …, Xn); {Còn lại một mạch thì đưa thẳng mạch đó sang miền y}
  end;

begin {Thuật toán sắp xếp trộn}
  Flag := True;
  len := 1;
  while len < n do
    begin
      if Flag then MergeByLength(k, t, len) else MergeByLength(t, k, len);
      len := len * 2;
      Flag := not Flag; {Đảo cờ để luân phiên vai trò của k và t}
    end;
  if not Flag then k := t; {Nếu kết quả cuối cùng đang nằm trong t thì sao chép kết quả vào k}
end;
```

Về độ phức tạp của thuật toán, ta thấy rằng trong thủ tục Merge, phép toán tích cực là thao tác đưa một khoá vào miền sắp xếp. Mỗi lần gọi thủ tục MergeByLength, tất cả các phần tử trong dãy khoá được chuyển hoàn toàn sang miền sắp xếp, nên độ phức tạp của thủ tục MergeByLength là O(n). Thủ tục MergeSort có vòng lặp thực hiện không quá log2n + 1 lời gọi MergeByLength bởi biến len sẽ được tăng theo cấp số nhân công bội 2. Từ đó suy ra **độ phức tạp của MergeSort là O(nlog$_2$n)** bất chấp trạng thái dữ liệu vào.

Cùng là những thuật toán sắp xếp tổng quát với độ phức tạp trung bình như nhau, nhưng không giống như QuickSort hay HeapSort, MergeSort có tính **ổn định**. Nhược điểm của MergeSort là nó phải dùng thêm một vùng nhớ để chứa dãy khoá phụ có kích thước bằng dãy khoá ban đầu.

Người ta còn có thể lợi dụng được trạng thái dữ liệu vào để khiến MergeSort chạy nhanh hơn: ngay từ đầu, ta không coi mỗi phần tử của dãy khoá là một mạch mà coi những đoạn đã được sắp trong dãy khoá là một mạch. Bởi một dãy khoá bất kỳ có thể coi là gồm các mạch đã sắp xếp nằm liên tiếp nhau. Khi đó người ta gọi phương pháp này là phương pháp **trộn hai đường tự nhiên.**

Tổng quát hơn nữa, thay vì phép trộn hai mạch, người ta có thể sử dụng phép trộn k mạch, khi đó ta được thuật toán sắp xếp trộn k đường.

8.12. CÀI ĐẶT

Ta sẽ cài đặt tất cả các thuật toán sắp xếp nêu trên, với dữ liệu vào được đặt trong file văn bản SORT.INP chứa không nhiều hơn 15000 khoá và giá trị mỗi khoá là số tự nhiên không quá 15000. Kết quả được ghi ra file văn bản SORT.OUT chứa dãy khoá được sắp, mỗi khoá trên một dòng.

SORT.INP	SORT.OUT
1 4 3 2 5 7 9 8 10 6	1 2 3 4 5 6 7 8 9 10

Chương trình có giao diện dưới dạng menu, mỗi chức năng tương ứng với một thuật toán sắp xếp. Tại mỗi thuật toán sắp xếp, ta thêm một vài lệnh đo thời gian thực tế của nó (chỉ đo thời gian thực hiện giải thuật, không tính thời gian nhập liệu và in kết quả).

Ở thuật toán sắp xếp bằng cơ số theo cách hoán vị phần tử, ta chọn hệ nhị phân. Ở thuật toán sắp xếp bằng cơ số trực tiếp, ta sử dụng hệ cơ số 256, khi đó một giá trị số tự nhiên $x \le 15000$ sẽ được biểu diễn bằng hai chữ số trong hệ 256:

Chữ số hàng đơn vị là x mod 256 = x mod 2^8 = x and 255 = x and $FF;

Chữ số còn lại (= chữ số ở hàng cao nhất) là x div 256 = x div 2^8 = x shr 8;

P_2_08_1.PAS * Các thuật toán sắp xếp

```
{$M 65520 0 655360}
program SortingAlgorithmsDemo;
uses crt;
const
  InputFile  = 'SORT.INP';
  OutputFile = 'SORT.OUT';
```

```
  max = 15000;
  maxV = 15000;
  Interval = 1193180 / 65536; {Tần số đồng hồ ≈ 18.2 lần / giây}
  nMenu = 12;
  SMenu: array[0..nMenu] of String =
  (
    '  0. Display Input',
    '  1. SelectionSort',
    '  2. BubbleSort',
    '  3. InsertionSort',
    '  4. InsertionSort with binary searching',
    '  5. ShellSort',
    '  6. QuickSort',
    '  7. HeapSort',
    '  8. Distribution Counting',
    '  9. Exchange RadixSort',
    ' 10. Straight RadixSort',
    ' 11. MergeSort',
    ' 12. Exit'
  );
type
  TArr = array[1..max] of Integer;
  TCount = array[0..maxV] of Integer;
var
  k: TArr;
  n: Integer;
  selected: Integer;
  StTime: LongInt;
  Time: LongInt absolute 0:$46C; {Biến đếm nhịp đồng hồ}

procedure Enter; {Trước mỗi thuật toán sắp xếp, gọi thủ tục này để nhập liệu}
var
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  n := 0;
  while not SeekEof(f) do
    begin
      Inc(n); Read(f, k[n]);
    end;
  Close(f);
  StTime := Time; {Nhập xong bắt đầu tính thời gian ngay}
end;

procedure PrintInput; {In dữ liệu}
var
  i: Integer;
begin
  Enter;
  for i := 1 to n do Write(k[i]:8);
  Write('Press any key to return to menu…');
  ReadKey
end;

procedure PrintResult; {In kết quả của mỗi thuật toán sắp xếp}
var
  f: Text;
  i: Integer;
  ch: Char;
begin
  {Trước hết in ra thời gian thực thi}
  WriteLn('Running Time = ', (Time - StTime) / Interval:1:10, ' (s)');
  Assign(f, OutputFile); Rewrite(f);
  for i := 1 to n do WriteLn(f, k[i]);
```

```
  Close(f);
  Write('Press <P> to print Output, another key to return to menu…');
  ch := ReadKey; WriteLn(ch);
  if Upcase(ch) = 'P' then
    begin
      for i := 1 to n do Write(k[i]:8);
      WriteLn;
      Write('Press any key to return to menu…');
      ReadKey;
    end;
end;

procedure Swap(var x, y: Integer); {Thủ tục đảo giá trị hai tham biến x, y}
var
  t: Integer;
begin
  t := x; x := y; y := t;
end;

(** SELECTIONSORT *************************************************)
procedure SelectionSort;
var
  i, j, jmin: Integer;
begin
  Enter;
  for i := 1 to n - 1 do
    begin
      jmin := i;
      for j := i + 1 to n do
        if k[j] < k[jmin] then jmin := j;
      if jmin <> i then Swap(k[i], k[jmin]);
    end;
  PrintResult;
end;

(** BUBBLESORT ****************************************************)
procedure BubbleSort;
var
  i, j: Integer;
begin
  Enter;
  for i := 2 to n do
    for j := n downto i do
      if k[j - 1] > k[j] then Swap(k[j - 1], k[j]);
  PrintResult;
end;

(** INSERTIONSORT *************************************************)
procedure InsertionSort;
var
  i, j, tmp: Integer;
begin
  Enter;
  for i := 2 to n do
    begin
      tmp := k[i]; j := i - 1;
      while (j > 0) and (tmp < k[j]) do
        begin
          k[j + 1] := k[j];
          Dec(j);
        end;
      k[j + 1] := tmp;
    end;
  PrintResult;
```

```
end;

(** INSERTIONSORT WITH BINARY SEARCHING ***************************)
procedure AdvancedInsertionSort;
var
  i, inf, sup, median, tmp: Integer;
begin
  Enter;
  for i := 2 to n do
    begin
      tmp := k[i];
      inf := 1; sup := i - 1;
      repeat
        median := (inf + sup) shr 1;
        if tmp < k[median] then sup := median - 1
        else inf := median + 1;
      until inf > sup;
      Move(k[inf], k[inf + 1], (i - inf) * SizeOf(k[1]));
      k[inf] := tmp;
    end;
  PrintResult;
end;

(** SHELLSORT ****************************************************)
procedure ShellSort;
var
  tmp: Integer;
  i, j, h: Integer;
begin
  Enter;
  h := n shr 1;
  while h <> 0 do
    begin
      for i := h + 1 to n do
        begin
          tmp := k[i]; j := i - h;
          while (j > 0) and (k[j] > tmp) do
            begin
              k[j + h] := k[j];
              j := j - h;
            end;
          k[j + h] := tmp;
        end;
      h := h shr 1;
    end;
  PrintResult;
end;

(** QUICKSORT ****************************************************)
procedure QuickSort;

  procedure Partition(L, H: Integer);
  var
    i, j: Integer;
    Pivot: Integer;
  begin
    if L >= H then Exit;
    Pivot := k[L + Random(H - L + 1)];
    i := L; j := H;
    repeat
      while k[i] < Pivot do Inc(i);
      while k[j] > Pivot do Dec(j);
      if i <= j then
        begin
```

```
        if i < j then Swap(k[i], k[j]);
        Inc(i); Dec(j);
      end;
    until i > j;
    Partition(L, j); Partition(i, H);
  end;

begin
  Enter;
  Partition(1, n);
  PrintResult;
end;

(** HEAPSORT *****************************************************)
procedure HeapSort;
var
  r, i: Integer;

  procedure Adjust(root, endnode: Integer);
  var
    key, c: Integer;
  begin
    key := k[root];
    while root shl 1 <= endnode do
      begin
        c := root shl 1;
        if (c < endnode) and (k[c] < k[c + 1]) then Inc(c);
        if k[c] <= key then Break;
        k[root] := k[c]; root := c;
      end;
    k[root] := key;
  end;

begin
  Enter;
  for r := n shr 1 downto 1 do Adjust(r, n);
  for i := n downto 2 do
    begin
      Swap(k[1], k[i]);
      Adjust(1, i - 1);
    end;
  PrintResult;
end;

(** DISTRIBUTION COUNTING ****************************************)
procedure DistributionCounting;
var
  x: TArr;
  c: TCount;
  i, V: Integer;
begin
  Enter;
  FillChar(c, SizeOf(c), 0);
  for i := 1 to n do Inc(c[k[i]]);
  for V := 1 to MaxV do c[V] := c[V - 1] + c[V];
  for i := n downto 1 do
    begin
      V := k[i];
      x[c[V]] := k[i];
      Dec(c[V]);
    end;
  k := x;
  PrintResult;
end;
```

```
(** EXCHANGE RADIXSORT *********************************************)
procedure RadixSort;
const
  MaxBit = 13;
var
  MaskBit: array[0..MaxBit] of Integer;
  MaxValue, i: Integer;

  procedure Partition(L, H, BIndex: Integer);
  var
    i, j, Mask: Integer;
  begin
    if L >= H then Exit;
    i := L; j := H; Mask := MaskBit[BIndex];
    repeat
      while (i < j) and (k[i] and Mask = 0) do Inc(i);
      while (i < j) and (k[j] and Mask <> 0) do Dec(j);
      Swap(k[i], k[j]);
    until i = j;
    if k[j] and Mask = 0 then Inc(j);
    if BIndex > 0 then
      begin
        Partition(L, j - 1, BIndex - 1); Partition(j, H, BIndex - 1);
      end;
  end;

begin
  Enter;
  for i := 0 to MaxBit do MaskBit[i] := 1 shl i;
  maxValue := k[1];
  for i := 2 to n do
    if k[i] > MaxValue then maxValue := k[i];
  i := 0;
  while (i < MaxBit) and (MaskBit[i + 1] <= MaxValue) do Inc(i);
  Partition(1, n, i);
  PrintResult;
end;

(** STRAIGHT RADIXSORT *********************************************)
procedure StraightRadixSort;
const
  Radix = 256;
  nDigit = 2;
var
  t: TArr;
  p: Integer;
  Flag: Boolean;

  function GetDigit(key, p: Integer): Integer;
  begin
    if p = 0 then GetDigit := key and $FF
    else GetDigit := key shr 8;
  end;

  procedure DCount(var x, y: TArr; p: Integer);
  var
    c: array[0..Radix - 1] of Integer;
    i, d: Integer;
  begin
    FillChar(c, SizeOf(c), 0);
    for i := 1 to n do
      begin
        d := GetDigit(x[i], p); Inc(c[d]);
```

```
      end;
    for d := 1 to Radix - 1 do c[d] := c[d - 1] + c[d];
    for i := n downto 1 do
      begin
        d := GetDigit(x[i], p);
        y[c[d]] := x[i];
        Dec(c[d]);
      end;
  end;

begin
  Enter;
  Flag := True;
  for p := 0 to nDigit - 1 do
    begin
      if Flag then DCount(k, t, p)
      else DCount(t, k, p);
      Flag := not Flag;
    end;
  if not Flag then k := t;
  PrintResult;
end;

(** MERGESORT ****************************************************)
procedure MergeSort;
var
  t: TArr;
  Flag: Boolean;
  len: Integer;

  procedure Merge(var Source, Dest: TArr; a, b, c: Integer);
  var
    i, j, p: Integer;
  begin
    p := a; i := a; j := b + 1;
    while (i <= b) and (j <= c) do
      begin
        if Source[i] <= Source[j] then
          begin
            Dest[p] := Source[i]; Inc(i);
          end
        else
          begin
            Dest[p] := Source[j]; Inc(j);
          end;
        Inc(p);
      end;
    if i <= b then
      Move(Source[i], Dest[p], (b - i + 1) * SizeOf(Source[1]))
    else
      Move(Source[j], Dest[p], (c - j + 1) * SizeOf(Source[1]));
  end;

  procedure MergeByLength(var Source, Dest: TArr; len: Integer);
  var
    a, b, c: Integer;
  begin
    a := 1; b := len; c := len shl 1;
    while c <= n do
      begin
        Merge(Source, Dest, a, b, c);
        a := a + len shl 1; b := b + len shl 1; c := c + len shl 1;
      end;
    if b < n then Merge(Source, Dest, a, b, n)
```

```
    else
      Move(Source[a], Dest[a], (n - a + 1) * SizeOf(Source[1]));
  end;

begin
  Enter;
  len := 1; Flag := True;
  FillChar(t, SizeOf(t), 0);
  while len < n do
    begin
      if Flag then MergeByLength(k, t, len)
      else MergeByLength(t, k, len);
      len := len shl 1;
      Flag := not Flag;
    end;
  if not Flag then k := t;
  PrintResult;
end;
(******************************************************************)

function MenuSelect: Integer;
var
  ch: Integer;
begin
  Clrscr;
  WriteLn('Sorting Algorithms Demos; Input: SORT.INP; Output: SORT.OUT');
  for ch := 0 to nMenu do WriteLn(SMenu[ch]);
  Write('Enter your choice: '); ReadLn(ch);
  MenuSelect := ch;
end;

begin
  repeat
    selected := MenuSelect;
    WriteLn(SMenu[selected]);
    case selected of
      0: PrintInput;
      1: SelectionSort;
      2: BubbleSort;
      3: InsertionSort;
      4: AdvancedInsertionSort;
      5: ShellSort;
      6: QuickSort;
      7: HeapSort;
      8: DistributionCounting;
      9: RadixSort;
      10: StraightRadixSort;
      11: MergeSort;
      12: Halt;
    end;
  until False;
end.
```

8.13. ĐÁNH GIÁ, NHẬN XÉT

Những con số về thời gian và tốc độ chương trình đo được là qua thử nghiệm trên một bộ dữ liệu cụ thể, với một máy tính cụ thể và một công cụ lập trình cụ thể. Với bộ dữ liệu khác, máy tính và công cụ lập trình khác, kết quả có thể khác. Tuy vậy, việc đo thời gian thực thi của từng thuật toán sắp xếp vẫn cần thiết nếu ta muốn so sánh tốc độ của các thuật toán cùng cấp phức tạp bởi các tính toán trên lý thuyết đôi khi bị lệch so với thực tế vì nhiều lý do khác nhau.

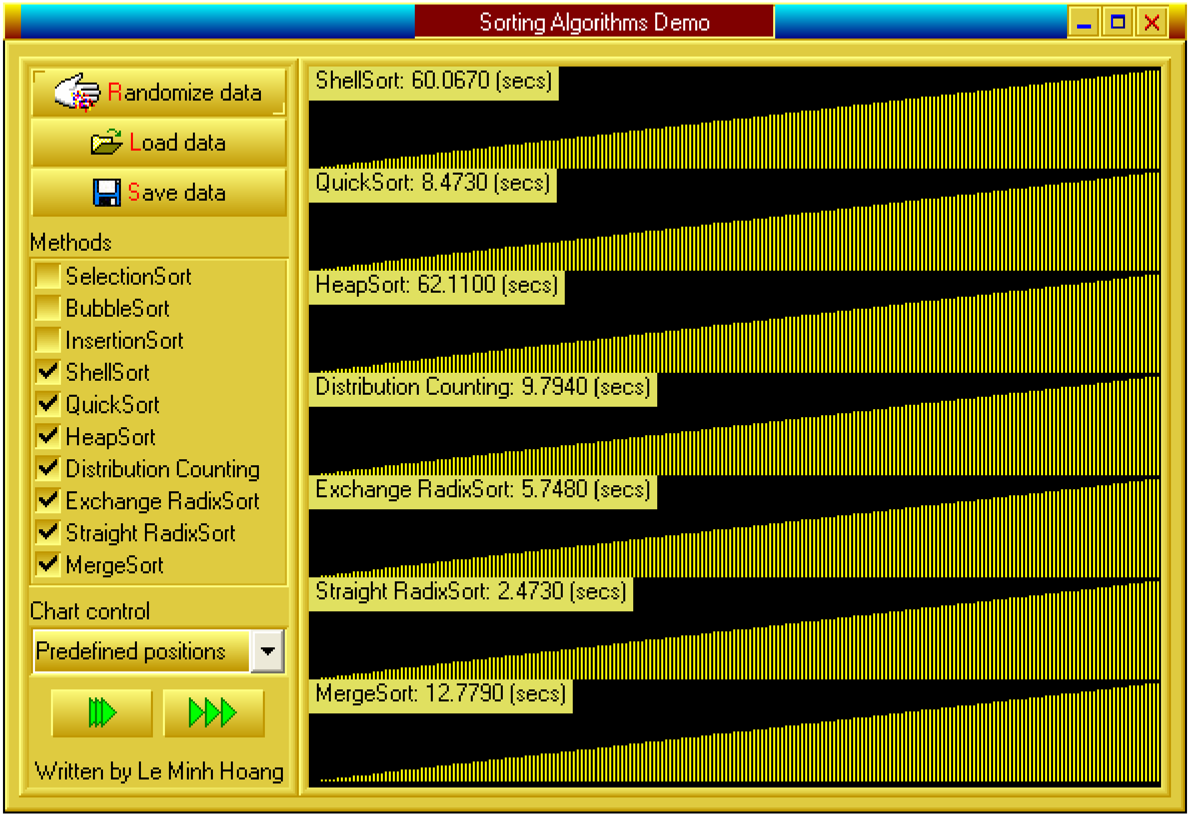
Có một vấn đề đặt ra là ngoài những thuật toán sắp xếp cấp $O(n^2)$, rất khó có thể đo được tốc độ trung bình của những thuật toán sắp xếp còn lại khi mà chúng đều chạy không tới một nhịp đồng hồ thời gian thực (đều cho thời gian chạy bằng 0 do không kịp đo thời gian). Một cách giải quyết là cho mỗi thuật toán QuickSort, RadixSort, … thực hiện c lần (c là một số nguyên đủ lớn) trên các bộ dữ liệu ngẫu nhiên rồi lấy thời gian tổng chia cho c, hay có thể tăng kích thước dữ liệu (điều này có thể dẫn đến việc phải sửa lại một vài chỗ trong chương trình hoặc thậm chí phải thay đổi môi trường lập trình).

Tôi đã viết lại chương trình này trên Borland Delphi để đưa vào một số cải tiến:

- Có thể chạy với kích thước dữ liệu lớn hơn rất nhiều (hàng triệu khóa)
- Thiết kế dựa trên kiến trúc đa luồng (MultiThreads) cho phép chạy đồng thời () hai hay nhiều thuật toán sắp xếp để so sánh tốc độ, hiển thị quá trình sắp xếp trực quan trên màn hình.
- Cũng cho phép chạy tuần tự () các thuật toán sắp xếp để đo thời gian thực hiện chính xác của chúng.

Chú ý: Để chương trình không bị ảnh hưởng bởi các phần mềm khác đang chạy, khi bấm  hoặc  khởi động các threads, bàn phím, chuột và tất cả các phần mềm khác sẽ bị treo tạm thời đến khi các threads thực hiện xong. Vì vậy không nên chạy các thuật toán sắp xếp chậm với dữ liệu lớn, sẽ không thể đợi đến khi các threads kết thúc và sẽ phải tắt máy khởi động lại.

Hình 36 là giao diện của chương trình, bạn có thể tham khảo mã nguồn chương trình kèm theo:



Hình 36: Cài đặt các thuật toán sắp xếp với dữ liệu lớn

Cùng một mục đích sắp xếp như nhau, nhưng có nhiều phương pháp giải quyết khác nhau. Nếu chỉ dựa vào thời gian đo được trong một ví dụ cụ thể mà đánh giá thuật toán này tốt hơn thuật toán kia về mọi mặt là điều không nên. Việc chọn một thuật toán sắp xếp thích hợp cho phù hợp với từng yêu cầu, từng điều kiện cụ thể là kỹ năng của người lập trình.

Những thuật toán có độ phức tạp $O(n^2)$ thì chỉ nên áp dụng trong chương trình có ít lần sắp xếp và với kích thước n nhỏ. Về tốc độ, BubbleSort luôn luôn đứng bét, nhưng mã lệnh của nó lại hết sức đơn giản mà người mới học lập trình nào cũng có thể cài đặt được, tính ổn định của BubbleSort cũng rất đáng chú ý. Trong những thuật toán có độ phức tạp $O(n^2)$, InsertionSort tỏ ra nhanh hơn những phương pháp còn lại và cũng có tính ổn định, mã lệnh cũng tương đối đơn giản, dễ nhớ. SelectionSort thì không ổn định nhưng với n nhỏ, việc chọn ra m phần tử nhỏ nhất có thể thực hiện dễ dàng chứ không cần phải sắp xếp lại toàn bộ như sắp xếp chèn.

Thuật toán đếm phân phối và thuật toán sắp xếp bằng cơ số nên được tận dụng trong trường hợp các khoá sắp xếp là số tự nhiên (hay là một kiểu dữ liệu có thể quy ra thành các số tự nhiên) bởi những thuật toán này có tốc độ rất cao. Thuật toán sắp xếp bằng cơ số cũng có thể sắp xếp dãy khoá có số thực hay số âm nhưng ta phải biết được cách thức lưu trữ các kiểu dữ liệu đó trên máy tính thì mới có thể làm được.

QuickSort, HeapSort, MergeSort và ShellSort là những thuật toán sắp xếp tổng quát, dãy khoá thuộc kiểu dữ liệu có thứ tự nào cũng có thể áp dụng được chứ không nhất thiết phải là các số.

QuickSort gặp nhược điểm trong trường hợp suy biến nhưng xác suất xảy ra trường hợp này rất nhỏ. HeapSort thì mã lệnh hơi phức tạp và khó nhớ, nhưng nếu cần chọn ra m phần tử lớn nhất trong dãy khoá thì dùng HeapSort sẽ không phải sắp xếp lại toàn bộ dãy. MergeSort phải đòi hỏi thêm một không gian nhớ phụ, nên áp dụng nó trong trường hợp sắp xếp trên file. Còn ShellSort thì hơi khó trong việc đánh giá về thời gian thực thi, nó là sửa đổi của thuật toán sắp xếp chèn nhưng lại có tốc độ tốt, mã lệnh đơn giản và lượng bộ nhớ cần huy động rất ít. Tuy nhiên, những nhược điểm của bốn phương pháp này quá nhỏ so với ưu điểm chung của chúng là **nhanh**. Hơn nữa, chúng được đánh giá cao không chỉ vì tính tổng quát và tốc độ nhanh, mà còn là kết quả của những cách tiếp cận khoa học đối với bài toán sắp xếp.

Những thuật toán trên không chỉ đơn thuần là cho ta hiểu thêm về một cách sắp xếp mới, mà kỹ thuật cài đặt chúng (với mã lệnh tối ưu) cũng dạy cho chúng ta nhiều điều: Kỹ thuật sử dụng số ngẫu nhiên, kỹ thuật "chia để trị", kỹ thuật dùng các biến với vai trò luân phiên v.v…Vậy nên nắm vững nội dung của những thuật toán đó, mà cách thuộc tốt nhất chính là cài đặt chúng vài lần với các ràng buộc dữ liệu khác nhau (nếu có thể thử được trên hai ngôn ngữ lập trình thì rất tốt) và cũng đừng quên kỹ thuật sắp xếp bằng chỉ số.

Bài tập

Bài 1

Viết thuật toán QuickSort không đệ quy

Bài 2

Hãy viết những thuật toán sắp xếp nêu trên với danh sách những xâu ký tự gồm 3 chữ cái thường, để sắp xếp chúng theo thứ tự từ điển.

Bài 3

Hãy viết lại tất cả những thuật toán nêu trên với phương pháp sắp xếp bằng chỉ số trên một dãy số cần sắp không tăng (giảm dần).

Bài 5

Cho một danh sách thí sinh gồm n người, mỗi người cho biết tên và điểm thi, hãy chọn ra m người điểm cao nhất. Giải quyết bằng thuật toán có độ phức tạp tính toán trung bình O(n)

Bài 6

Thuật toán sắp xếp bằng cơ số trực tiếp có ổn định không ? Tại sao ?

Bài 7

Cài đặt thuật toán sắp xếp trộn hai đường tự nhiên

Bài 8

Tìm hiểu phép trộn k đường và các phương pháp sắp xếp ngoài (trên tệp truy nhập tuần tự và tệp truy nhập ngẫu nhiên)

§9. TÌM KIẾM (SEARCHING)

9.1. BÀI TOÁN TÌM KIẾM

Cùng với sắp xếp, tìm kiếm là một đòi hỏi rất thường xuyên trong các ứng dụng tin học. Bài toán tìm kiếm có thể phát biểu như sau:

Cho một dãy gồm n bản ghi $r_1, r_2, …, r_n$. Mỗi bản ghi r_i ($1 \le i \le n$) tương ứng với một khoá k_i. Hãy tìm bản ghi có giá trị khoá bằng X cho trước.

X được gọi là khoá tìm kiếm hay đối trị tìm kiếm (argument).

Công việc tìm kiếm sẽ hoàn thành nếu như có một trong hai tình huống sau xảy ra:

Tìm được bản ghi có khoá tương ứng bằng X, lúc đó phép tìm kiếm thành công (successful).

Không tìm được bản ghi nào có khoá tìm kiếm bằng X cả, phép tìm kiếm thất bại (unsuccessful).

Tương tự như sắp xếp, ta coi khoá của một bản ghi là đại diện cho bản ghi đó. Và trong một số thuật toán sẽ trình bày dưới đây, ta coi kiểu dữ liệu cho mỗi khoá cũng có tên gọi là TKey.

```
const
  n = …; {Số khoá trong dãy khoá, có thể khai dưới dạng biến số nguyên để tuỳ biến hơn}
type
  TKey = …; {Kiểu dữ liệu một khoá}
  TArray = array[1..n] of TKey;
var
  k: TArray; {Dãy khoá}
```

9.2. TÌM KIẾM TUẦN TỰ (SEQUENTIAL SEARCH)

Tìm kiếm tuần tự là một kỹ thuật tìm kiếm đơn giản. Nội dung của nó như sau: Bắt đầu từ bản ghi đầu tiên, lần lượt so sánh khoá tìm kiếm với khoá tương ứng của các bản ghi trong danh sách, cho tới khi tìm thấy bản ghi mong muốn hoặc đã duyệt hết danh sách mà chưa thấy

```
{Tìm kiếm tuần tự trên dãy khoá k1, k2, …, kn; hàm này thử tìm xem trong dãy có khoá nào = X không, nếu thấy nó trả về chỉ
số của khoá ấy, nếu không thấy nó trả về 0. Có sử dụng một khoá phụ kn+1 được gán giá trị = X}
function SequentialSearch(X: TKey): Integer;
var
  i: Integer;
begin
  i := 1;
  while (i <= n) and (ki ≠ X) do i := i + 1;
  if i = n + 1 then SequentialSearch := 0
  else SequentialSearch := i;
end;
```

Dễ thấy rằng độ phức tạp của thuật toán tìm kiếm tuần tự trong trường hợp tốt nhất là O(1), trong trường hợp xấu nhất là O(n) và trong trường hợp trung bình cũng là O(n).

9.3. TÌM KIẾM NHỊ PHÂN (BINARY SEARCH)

Phép tìm kiếm nhị phân có thể áp dụng trên dãy khoá đã có thứ tự: $k_1 \le k_2 \le … \le k_n$.

Giả sử ta cần tìm trong đoạn k_{inf}, k_{inf+1}, …, k_{sup} với khoá tìm kiếm là X, trước hết ta xét khoá nằm giữa dãy k_{median} với median = (inf + sup) div 2;

Nếu k_{median} < X thì có nghĩa là đoạn từ k_{inf} tới k_{median} chỉ chứa toàn khoá < X, ta tiến hành tìm kiếm tiếp với đoạn từ $k_{median + 1}$ tới k_{sup}.

Nếu k_{median} > X thì có nghĩa là đoạn từ k_{median} tới k_{sup} chỉ chứa toàn khoá > X, ta tiến hành tìm kiếm tiếp với đoạn từ k_{inf} tới $k_{median - 1}$.

Nếu k_{median} = X thì việc tìm kiếm thành công (kết thúc quá trình tìm kiếm).

Quá trình tìm kiếm sẽ thất bại nếu đến một bước nào đó, đoạn tìm kiếm là rỗng (inf > sup).

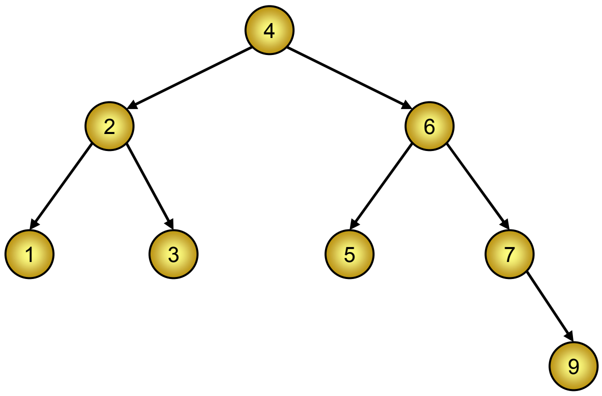
```
{Tìm kiếm nhị phân trên dãy khoá k1 ≤ k2 ≤ … ≤ kn; hàm này thử tìm xem trong dãy có khoá nào = X không, nếu thấy nó trả về
chỉ số của khoá ấy, nếu không thấy nó trả về 0}
function BinarySearch(X: TKey): Integer;
var
  inf, sup, median: Integer;
begin
  inf := 1; sup := n;
  while inf ≤ sup do
    begin
      median := (inf + sup) div 2;
      if kmedian = X then
        begin
          BinarySearch := median;
          Exit;
        end;
      if kmedian < X then inf := median + 1
      else sup := median - 1;
    end;
  BinarySearch := 0;
end;
```

Người ta đã chứng minh được độ phức tạp tính toán của thuật toán tìm kiếm nhị phân trong trường hợp tốt nhất là O(1), trong trường hợp xấu nhất là $O(\log_2 n)$ và trong trường hợp trung bình cũng là $O(\log_2 n)$. Tuy nhiên, ta không nên quên rằng trước khi sử dụng tìm kiếm nhị phân, dãy khoá phải được sắp xếp rồi, tức là thời gian chi phí cho việc sắp xếp cũng phải tính đến. Nếu dãy khoá luôn luôn biến động bởi phép bổ sung hay loại bớt đi thì lúc đó chi phí cho sắp xếp lại nổi lên rất rõ làm bộc lộ nhược điểm của phương pháp này.

9.4. CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE - BST)

Cho n khoá k_1, k_2, …, k_n, trên các khoá có quan hệ thứ tự toàn phần. Cây nhị phân tìm kiếm ứng với dãy khoá đó là một cây nhị phân mà mỗi nút chứa giá trị một khoá trong n khoá đã cho, hai giá trị chứa trong hai nút bất kỳ là khác nhau. Đối với mọi nút trên cây, tính chất sau luôn được thoả mãn:

- Mọi khoá nằm trong cây con trái của nút đó đều nhỏ hơn khoá ứng với nút đó.
- Mọi khoá nằm trong cây con phải của nút đó đều lớn hơn khoá ứng với nút đó



Hình 37: Cây nhị phân tìm kiếm

Thuật toán tìm kiếm trên cây có thể mô tả chung như sau:

Trước hết, khoá tìm kiếm X được so sánh với khoá ở gốc cây, và 4 tình huống có thể xảy ra:

Không có gốc (cây rỗng): X không có trên cây, phép tìm kiếm thất bại

X trùng với khoá ở gốc: Phép tìm kiếm thành công

X nhỏ hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con trái của gốc với cách làm tương tự

X lớn hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con phải của gốc với cách làm tương tự

Giả sử cấu trúc một nút của cây được mô tả như sau:

```
type
  PNode = ^TNode; {Con trỏ chứa liên kết tới một nút}
  TNode = record {Cấu trúc nút}
    Info: TKey; {Trường chứa khoá}
    Left, Right: PNode; {con trỏ tới nút con trái và phải, trỏ tới nil nếu không có nút con trái (phải)}
  end;
Gốc của cây được lưu trong con trỏ Root. Cây rỗng thì Root = nil
```

Thuật toán tìm kiếm trên cây nhị phân tìm kiếm có thể viết như sau:

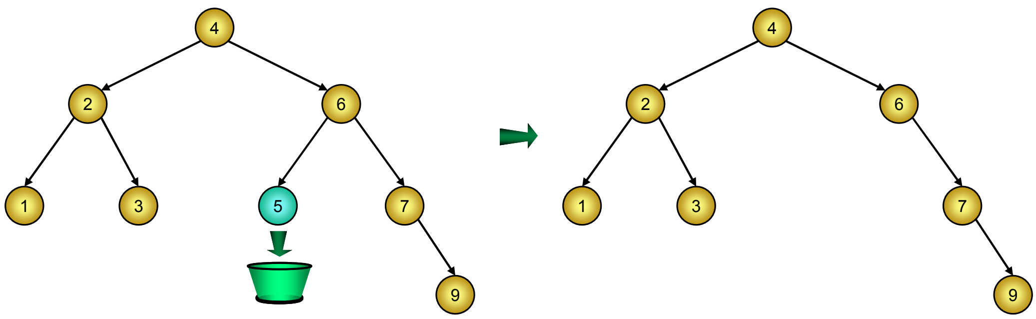
```
{Hàm tìm kiếm trên BST, nó trả về nút chứa khoá tìm kiếm X nếu tìm thấy, trả về nil nếu không tìm thấy}
function BSTSearch(X: TKey): PNode;
var
  p: PNode;
begin
  p := Root; {Bắt đầu với nút gốc}
  while p ≠ nil do
    if X = p^.Info then Break;
    else
      if X < p^.Info then p := p^.Left
      else p := p^.Right;
  BSTSearch := p;
end;
```

Thuật toán dựng cây nhị phân tìm kiếm từ dãy khoá k_1, k_2, …, k_n cũng được làm gần giống quá trình tìm kiếm. Ta chèn lần lượt các khoá vào cây, trước khi chèn, ta tìm xem khoá đó đã có trong cây hay chưa, nếu đã có rồi thì bỏ qua, nếu nó chưa có thì ta thêm nút mới chứa khoá cần chèn và nối nút đó vào cây nhị phân tìm kiếm.

```
{Thủ tục chèn khoá X vào BST}
procedure BSTInsert(X);
var
  p, q: PNode;
begin
  q := nil; p := Root; {Bắt đầu với p = nút gốc; q là con trỏ chạy đuổi theo sau}
  while p ≠ nil do
    begin
      q := p;
      if X = p^.Info then Break;
      else {X ≠  p^.Info thì cho p chạy sang nút con, q^ luôn giữ vai trò là cha của p^}
        if X < p^.Info then p := p^.Left
        else p := p^.Right;
    end;
  if p = nil then {Khoá X chưa có trong BST}
    begin
      New(p); {Tạo nút mới}
      p^.Info := X; {Đưa giá trị X vào nút mới tạo ra}
      p^.Left := nil; p^.Right := nil; {Nút mới khi chèn vào BST sẽ trở thành nút lá}
      if Root = nil then Root := NewNode {BST đang rỗng, đặt Root là nút mới tạo}
      else {Móc NewNode^ vào nút cha q^}
        if X < q^.Info then q^.Left := NewNode
        else q^.Right := NewNode;
    end;
end;
```

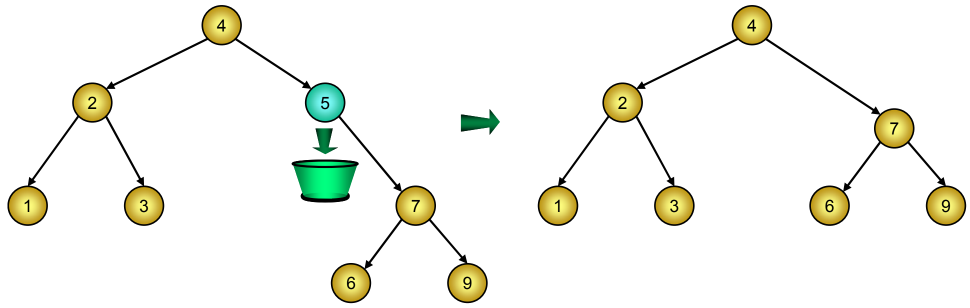
Phép loại bỏ trên cây nhị phân tìm kiếm không đơn giản như phép bổ sung hay phép tìm kiếm. Muốn xoá một giá trị trong cây nhị phân tìm kiếm (Tức là dựng lại cây mới chứa tất cả những giá trị còn lại), trước hết ta tìm xem giá trị cần xoá nằm ở nút D nào, có ba khả năng xảy ra:

- Nút D là nút lá, trường hợp này ta chỉ việc đem mối nối cũ trỏ tới nút D (từ nút cha của D) thay bởi nil, và giải phóng bộ nhớ cấp cho nút D (Hình 38).



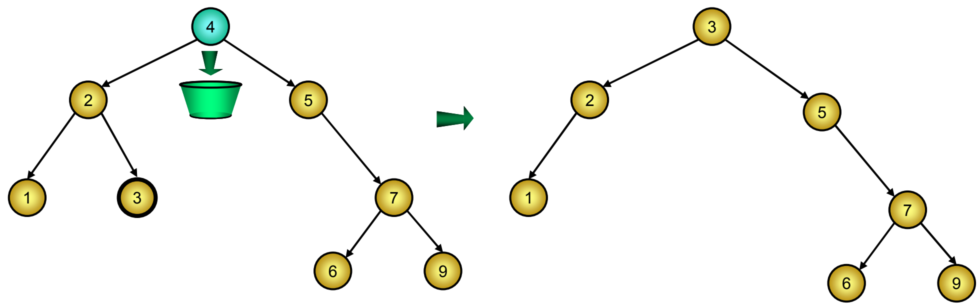
Hình 38: Xóa nút lá ở cây BST

- Nút D chỉ có một nhánh con, khi đó ta đem nút gốc của nhánh con đó thế vào chỗ nút D, tức là chỉnh lại mối nối: Từ nút cha của nút D không nối tới nút D nữa mà nối tới nhánh con duy nhất của nút D. Cuối cùng, ta giải phóng bộ nhớ đã cấp cho nút D (Hình 39)



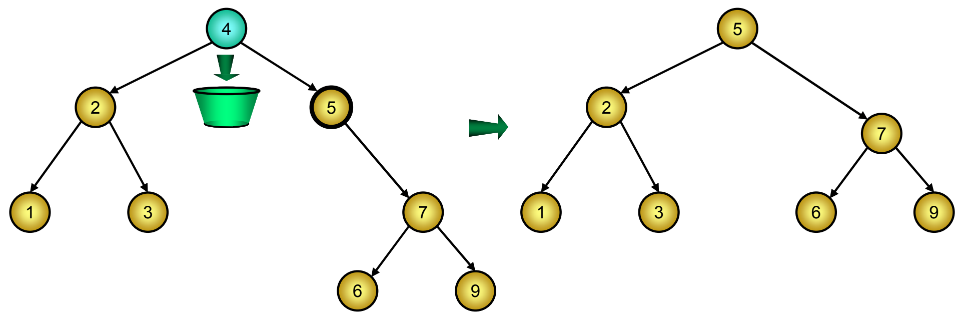
Hình 39. Xóa nút chỉ có một nhánh con trên cây BST

- Nút D có cả hai nhánh con trái và phải, khi đó có hai cách làm đều hợp lý cả:
  - Hoặc tìm nút chứa khoá lớn nhất trong cây con trái, đưa giá trị chứa trong đó sang nút D, rồi xoá nút này. Do tính chất của cây BST, nút chứa khoá lớn nhất trong cây con trái chính là nút cực phải của cây con trái nên nó không thể có hai con được, việc xoá đưa về hai trường hợp trên (Hình 40)



Hình 40: Xóa nút có cả hai nhánh con trên cây BST thay bằng nút cực phải của cây con trái

  - Hoặc tìm nút chứa khoá nhỏ nhất trong cây con phải, đưa giá trị chứa trong đó sang nút D, rồi xoá nút này. Do tính chất của cây BST, nút chứa khoá nhỏ nhất trong cây con phải chính là nút cực trái của cây con phải nên nó không thể có hai con được, việc xoá đưa về hai trường hợp trên.



Hình 41: Xóa nút có cả hai nhánh con trên cây BST thay bằng nút cực trái của cây con phải

```
{Thủ tục xoá khoá X khỏi BST}
procedure BSTDelete(X: TKey);
var
  p, q, Node, Child: PNode;
begin
  p := Root; q := nil; {Về sau, khi p trỏ sang nút khác, ta luôn giữ cho q^ luôn là cha của p^}
  while p ≠ nil do {Tìm xem trong cây có khoá X không?}
    begin
      if p^.Info = X then Break; {Tìm thấy}
      q := p;
      if X < p^.Info then p := p^.Left
      else p := p^.Right;
    end;
  if p = nil then Exit; {X không tồn tại trong BST nên không xoá được}
  if (p^.Left ≠ nil) and (p^.Right ≠ nil) then {p^ có cả con trái và con phải}
    begin
      Node := p; {Giữ lại nút chứa khoá X}
      q := p; p := p^.Left; {Chuyển sang nhánh con trái để tìm nút cực phải}
      while p^.Right ≠ nil do
        begin
          q := p; p := p^.Right;
        end;
      Node^.Info := p^.Info; {Chuyển giá trị từ nút cực phải trong nhánh con trái lên Node^}
    end;
  {Nút bị xoá giờ đây là nút p^, nó chỉ có nhiều nhất một con}
  {Nếu p^ có một nút con thì đem Child trỏ tới nút con đó, nếu không có thì Child = nil }
  if p^.Left ≠ nil then Child := p^.Left
  else Child := p^.Right;
  if p = Root then Root := Child; {Nút p^ bị xoá là gốc cây}
  else {Nút bị xoá p^ không phải gốc cây thì lấy mối nối từ cha của nó là q^ nối thẳng tới Child}
    if q^.Left = p then q^.Left := Child
    else q^.Right := Child;
  Dispose(p);
end;
```

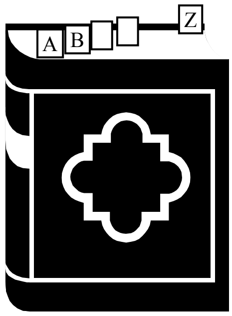
Trường hợp trung bình, thì các thao tác tìm kiếm, chèn, xoá trên BST có độ phức tạp là $O(\log_2 n)$. Còn trong trường hợp xấu nhất, cây nhị phân tìm kiếm bị suy biến thì các thao tác đó đều có độ phức tạp là O(n), với n là số nút trên cây BST.

Nếu ta mở rộng hơn khái niệm cây nhị phân tìm kiếm như sau: Giá trị lưu trong một nút lớn hơn **hoặc bằng** các giá trị lưu trong cây con trái và nhỏ hơn các giá trị lưu trong cây con phải. Thì chỉ cần sửa đổi thủ tục BSTInsert một chút, khi chèn lần lượt vào cây n giá trị, cây BST sẽ có n nút (có thể có hai nút chứa cùng một giá trị). Khi đó nếu ta duyệt các nút của cây theo kiểu trung thứ tự (inorder traversal), ta sẽ liệt kê được các giá trị lưu trong cây theo thứ tự tăng dần. Phương pháp sắp xếp này người ta gọi là Tree Sort. Độ phức tạp tính toán trung bình của Tree Sort là $O(n\log_2 n)$.

Phép tìm kiếm trên cây BST sẽ kém hiệu quả nếu như cây bị suy biến, người ta có nhiều cách xoay xở để tránh trường hợp này. Đó là phép quay cây để dựng cây nhị phân cân đối AVL, hay kỹ thuật dựng cây nhị phân tìm kiếm tối ưu. Những kỹ thuật này ta có thể tham khảo trong các tài liệu khác về cấu trúc dữ liệu và giải thuật.

9.5. PHÉP BĂM (HASH)

Tư tưởng của phép băm là dựa vào giá trị các khoá k1, k2, …, kn, chia các khoá đó ra thành các nhóm. **Những khoá thuộc cùng một nhóm có một đặc điểm chung** và đặc điểm này không có trong các nhóm khác. Khi có một khoá tìm kiếm X, trước hết ta xác định xem nếu X thuộc vào dãy khoá đã cho thì nó phải thuộc nhóm nào và tiến hành tìm kiếm trên nhóm đó.
Một ví dụ là trong cuốn từ điển, các bạn sinh viên thường dán vào 26 mảnh giấy nhỏ vào các trang để đánh dấu trang nào là trang khởi đầu của một đoạn chứa các từ có cùng chữ cái đầu. Để khi tra từ chỉ cần tìm trong các trang chứa những từ có cùng chữ cái đầu với từ cần tìm.



Một ví dụ khác là trên dãy các khoá số tự nhiên, ta có thể chia nó là làm m nhóm, mỗi nhóm gồm các khoá đồng dư theo mô-đun m.
Có nhiều cách cài đặt phép băm:
Cách thứ nhất là chia dãy khoá làm các đoạn, mỗi đoạn chứa những khoá thuộc cùng một nhóm và ghi nhận lại vị trí các đoạn đó. Để khi có khoá tìm kiếm, có thể xác định được ngay cần phải tìm khoá đó trong đoạn nào.
Cách thứ hai là chia dãy khoá làm m nhóm, Mỗi nhóm là một danh sách nối đơn chứa các giá trị khoá và ghi nhận lại chốt của mỗi danh sách nối đơn. Với một khoá tìm kiếm, ta xác định được phải tìm khoá đó trong danh sách nối đơn nào và tiến hành tìm kiếm tuần tự trên danh sách nối đơn đó. Với cách lưu trữ này, việc bổ sung cũng như loại bỏ một giá trị khỏi tập hợp khoá dễ dàng hơn rất nhiều phương pháp trên.
Cách thứ ba là nếu chia dãy khoá làm m nhóm, mỗi nhóm được lưu trữ dưới dạng cây nhị phân tìm kiếm và ghi nhận lại gốc của các cây nhị phân tìm kiếm đó, phương pháp này có thể nói là tốt hơn hai phương pháp trên, tuy nhiên dãy khoá phải có quan hệ thứ tự toàn phần thì mới làm được.

9.6. KHOÁ SỐ VỚI BÀI TOÁN TÌM KIẾM

Mọi dữ liệu lưu trữ trong máy tính đều được số hoá, tức là đều được lưu trữ bằng các đơn vị Bit, Byte, Word v.v… Điều đó có nghĩa là một giá trị khoá bất kỳ, ta hoàn toàn có thể biết được nó được mã hoá bằng con số như thế nào. Và một điều chắc chắn là hai khoá khác nhau sẽ được lưu trữ bằng hai số khác nhau.

Đối với bài toán sắp xếp, ta không thể đưa việc sắp xếp một dãy khoá bất kỳ về việc sắp xếp trên một dãy khoá số là mã của các khoá. Bởi quan hệ thứ tự trên các con số đó có thể khác với thứ tự cần sắp của các khoá.

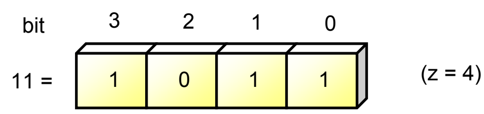
Nhưng đối với bài toán tìm kiếm thì khác, với một khoá tìm kiếm, Câu trả lời hoặc là "Không tìm thấy" hoặc là "Có tìm thấy và ở chỗ …" nên ta hoàn toàn có thể thay các khoá bằng các mã số của nó mà không bị sai lầm, chỉ lưu ý một điều là: hai khoá khác nhau phải mã hoá thành hai số khác nhau mà thôi.

Nói như vậy có nghĩa là việc nghiên cứu những thuật toán tìm kiếm trên các dãy khoá số rất quan trọng, và dưới đây ta sẽ trình bày một số phương pháp đó.

9.7. CÂY TÌM KIẾM SỐ HỌC (DIGITAL SEARCH TREE - DST)

Xét dãy khoá $k_1, k_2, \ldots, k_n$ là các số tự nhiên, mỗi giá trị khoá khi đổi ra hệ nhị phân có z chữ số nhị phân (bit), các bit này được đánh số từ 0 (là hàng đơn vị) tới z - 1 từ phải sang trái.

Ví dụ:

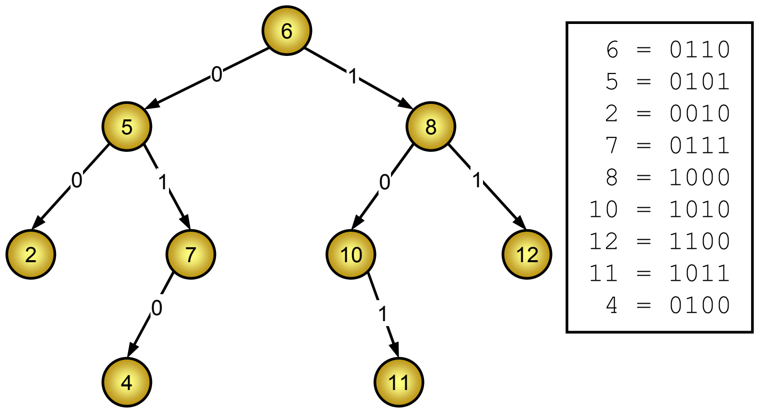


Hình 42: Đánh số các bit

Cây tìm kiếm số học chứa các giá trị khoá này có thể mô tả như sau: Trước hết, nó là một cây nhị phân mà mỗi nút chứa một giá trị khoá. Nút gốc có tối đa hai cây con, ngoài giá trị khoá chứa ở nút gốc, tất cả những giá trị khoá có bit cao nhất là 0 nằm trong cây con trái, còn tất cả những giá trị khoá có bit cao nhất là 1 nằm ở cây con phải. Đối với hai nút con của nút gốc, vấn đề tương tự đối với bit z - 2 (bit đứng thứ nhì từ trái sang).

So sánh cây tìm kiếm số học với cây nhị phân tìm kiếm, chúng chỉ khác nhau về cách chia hai cây con trái/phải. Đối với cây nhị phân tìm kiếm, việc chia này được thực hiện bằng cách so sánh với khoá nằm ở nút gốc, còn đối với cây tìm kiếm số học, nếu nút gốc có mức là d thì việc chia cây con được thực hiện theo bit thứ d tính từ trái sang (bit z - d) của mỗi khoá.

Ta nhận thấy rằng những khoá bắt đầu bằng bit 0 chắc chắn nhỏ hơn những khoá bắt đầu bằng bit 1, đó là điểm tương đồng giữa cây nhị phân tìm kiếm và cây tìm kiếm số học: Với mỗi nút nhánh: Mọi giá trị chứa trong cây con trái đều nhỏ hơn giá trị chứa trong cây con phải (Hình 43).



Hình 43: Cây tìm kiếm số học

Giả sử cấu trúc một nút của cây được mô tả như sau:

```
type
  PNode = ^TNode; {Con trỏ chứa liên kết tới một nút}
  TNode = record {Cấu trúc nút}
    Info: TKey; {Trường chứa khoá}
    Left, Right: PNode; {con trỏ tới nút con trái và phải, trỏ tới nil nếu không có nút con trái (phải)}
  end;
Gốc của cây được lưu trong con trỏ Root. Ban đầu nút Root = nil (cây rỗng)
```

Với khoá tìm kiếm X, việc tìm kiếm trên cây tìm kiếm số học có thể mô tả như sau: Ban đầu đứng ở nút gốc, xét lần lượt các bit của X từ trái sang phải (từ bit z - 1 tới bit 0), hễ gặp bit bằng 0 thì rẽ sang nút con trái, nếu gặp bit bằng 1 thì rẽ sang nút con phải. Quá trình cứ tiếp tục như vậy cho tới khi gặp một trong hai tình huống sau:

- Đi tới một nút rỗng (do rẽ theo một liên kết nil), quá trình tìm kiếm thất bại do khoá X không có trong cây.
- Đi tới một nút mang giá trị đúng bằng X, quá trình tìm kiếm thành công

```
{Hàm tìm kiếm trên cây tìm kiếm số học, nó trả về nút chứa khoá tìm kiếm X nếu tìm thấy, trả về nil nếu không tìm thấy. z là độ dài dãy bit biểu diễn một khoá}
function DSTSearch(X: TKey): PNode;
var
  b: Integer;
  p: PNode;
begin
  b := z; p := Root; {Bắt đầu với nút gốc}
  while (p ≠ nil) and (p^.Info ≠ X) do {Chưa gặp phải một trong 2 tình huống trên}
    begin
      b := b - 1; {Xét bit b của X}
      if <Bit b của X là 0> then p := p^.Left  {Gặp 0 rẽ trái}
      else p := p^.Right; {Gặp 1 rẽ phải}
    end;
  DSTSearch := p;
end;
```

Thuật toán dựng cây tìm kiếm số học từ dãy khoá $k_1, k_2, \ldots, k_n$ cũng được làm gần giống quá trình tìm kiếm. Ta chèn lần lượt các khoá vào cây, trước khi chèn, ta tìm xem khoá đó đã có trong cây hay chưa, nếu đã có rồi thì bỏ qua, nếu nó chưa có thì ta thêm nút mới chứa khoá cần chèn và nối nút đó vào cây tìm kiếm số học tại mối nối rỗng vừa rẽ sang khiến quá trình tìm kiếm thất bại

```
{Thủ tục chèn khoá X vào cây tìm kiếm số học}
procedure DSTInsert(X: TKey);
var
  b: Integer;
  p, q: PNode;
begin
  b := z;
  p := Root;
  while (p ≠ nil) and (p^.Info ≠ X) do
    begin
      b := b - 1; {Xét bit b của X}
      q := p; {Khi p chạy xuống nút con thì q^ luôn giữ vai trò là nút cha của p^}
      if <Bit b của X là 0> then p := p^.Left  {Gặp 0 rẽ trái}
      else p := p^.Right; {Gặp 1 rẽ phải}
    end;
  if p = nil then   {Giá trị X chưa có trong cây}
    begin
      New(p); {Tạo ra một nút mới p^}
      p^.Info := X; {Nút mới tạo ra sẽ chứa khoá X}
      p^.Left := nil; p^.Right := nil; {Nút mới đó sẽ trở thành một lá của cây}
      if Root = nil then Root := p  {Cây đang là rỗng thì nút mới thêm trở thành gốc}
      else {Không thì móc p^ vào mối nối vừa rẽ sang từ q^}
        if <Bit b của X là 0> then q^.Left := p
        else q^.Right := p;
    end;
end;
```

Muốn xoá bỏ một giá trị khỏi cây tìm kiếm số học, trước hết ta xác định nút chứa giá trị cần xoá là nút D nào, sau đó tìm trong nhánh cây gốc D ra một nút lá bất kỳ, chuyển giá trị chứa trong nút lá đó sang nút D rồi xoá nút lá.

```
{Thủ tục xoá khoá X khỏi cây tìm kiếm số học}
procedure DSTDelete(X: TKey);
var
  b: Integer;
  p, q, Node: PNode;
begin
  {Trước hết, tìm kiếm giá trị X xem nó nằm ở nút nào}
  b := z;
  p := Root;
  while (p ≠ nil) and (p^.Info ≠ X) do
    begin
      b := b - 1;
      q := p; {Mỗi lần p chuyển sang nút con, ta luôn đảm bảo cho q^ là nút cha của p^}
      if <Bit b của X là 0> then p := p^.Left
      else p := p^.Right;
    end;
  if p = nil then Exit; {X không tồn tại trong cây thì không xoá được}
  Node := p; {Giữ lại nút chứa khoá cần xoá}
  while (p^.Left ≠ nil) or (p^.Right ≠ nil) do {chừng nào p^ chưa phải là lá}
    begin
      q := p; {q chạy đuổi theo p, còn p chuyển xuống một trong 2 nhánh con}
      if p^.Left ≠ nil then p := p^.Left
      else p := p^.Right;
    end;
  Node^.Info := p^.Info; {Chuyển giá trị từ nút lá p^ sang nút Node^}
  if Root = p then Root := nil  {Cây chỉ gồm một nút gốc và bây giờ xoá cả gốc}
  else {Cắt mối nối từ q^ tới p^}
    if q^.Left = p then q^.Left := nil
    else q^.Right := nil;
  Dispose(p);
end;
```

Về mặt trung bình, các thao tác tìm kiếm, chèn, xoá trên cây tìm kiếm số học đều có độ phức tạp là $O(\log_2 n)$ còn trong trường hợp xấu nhất, độ phức tạp của các thao tác đó là O(z), bởi cây tìm kiếm số học có chiều cao không quá z + 1.

9.8. CÂY TÌM KIẾM CƠ SỐ (RADIX SEARCH TREE - RST)

Trong cây tìm kiếm số học, cũng như cây nhị phân tìm kiếm, phép tìm kiếm tại mỗi bước phải so sánh giá trị khoá X với giá trị lưu trong một nút của cây. Trong trường hợp các khoá có cấu trúc lớn, việc so sánh này có thể mất nhiều thời gian.

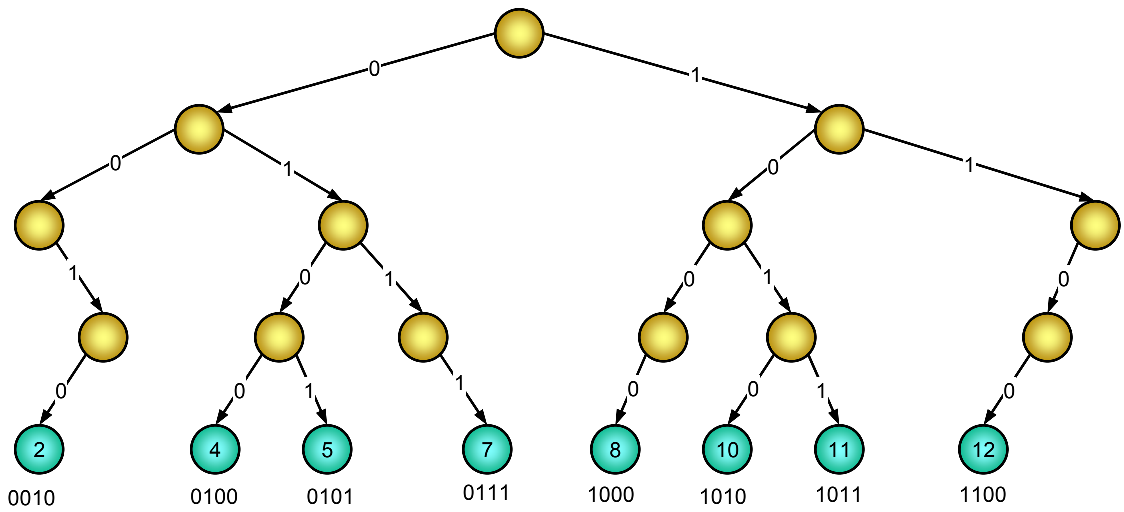
Cây tìm kiếm cơ số là một phương pháp khắc phục nhược điểm đó, nội dung của nó có thể tóm tắt như sau:

Trong cây tìm kiếm cơ số là một cây nhị phân, chỉ có nút lá chứa giá trị khoá, còn giá trị chứa trong các nút nhánh là vô nghĩa. Các nút lá của cây tìm kiếm cơ số đều nằm ở mức z + 1.

Đối với nút gốc của cây tìm kiếm cơ số, nó có tối đa hai nhánh con, mọi khoá chứa trong nút lá của nhánh con trái đều có bit cao nhất là 0, mọi khoá chứa trong nút lá của nhánh con phải đều có bit cao nhất là 1.

Đối với hai nhánh con của nút gốc, vấn đề tương tự với bit thứ z - 2, ví dụ với nhánh con trái của nút gốc, nó lại có tối đa hai nhánh con, mọi khoá chứa trong nút lá của nhánh con trái đều có bit thứ z - 2 là 0 (chúng bắt đầu bằng hai bit 00), mọi khoá chứa trong nút lá của nhánh con phải đều có bit thứ z - 2 là 1 (chúng bắt đầu bằng hai bit 01)…

Tổng quát với nút ở mức d, nó có tối đa hai nhánh con, mọi nút lá của nhánh con trái chứa khoá có bit z - d là 0, mọi nút lá của nhánh con phải chứa khoá có bit thứ z - d là 1 (Hình 44).



Hình 44: Cây tìm kiếm cơ số

Khác với cây nhị phân tìm kiếm hay cây tìm kiếm số học. Cây tìm kiếm cơ số được khởi tạo gồm có một nút gốc, và **nút gốc tồn tại trong suốt quá trình sử dụng**: nó không bao giờ bị xoá đi cả.

Để tìm kiếm một giá trị X trong cây tìm kiếm cơ số, ban đầu ta đứng ở nút gốc và duyệt dãy bit của X từ trái qua phải (từ bit z - 1 đến bit 0), gặp bit bằng 0 thì rẽ sang nút con trái còn gặp bit bằng 1 thì rẽ sang nút con phải, cứ tiếp tục như vậy cho tới khi một trong hai tình huống sau xảy ra:

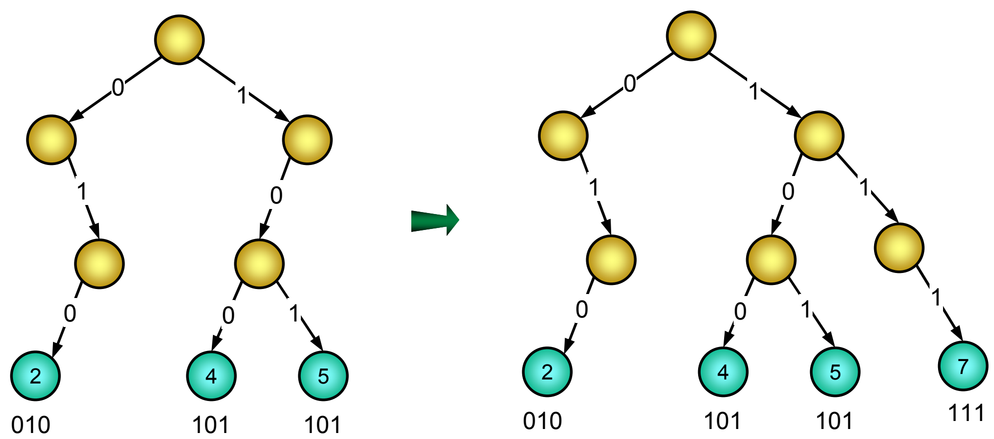
Hoặc đi tới một nút rỗng (do rẽ theo liên kết nil) quá trình tìm kiếm thất bại do X không có trong RST

Hoặc đã duyệt hết dãy bit của X và đang đứng ở một nút lá, quá trình tìm kiếm thành công vì chắc chắn nút lá đó chứa giá trị đúng bằng X.

```
{Hàm tìm kiếm trên cây tìm kiếm cơ số, nó trả về nút lá chứa khoá tìm kiếm X nếu tìm thấy, trả về nil nếu không tìm thấy.
z là độ dài dãy bit biểu diễn một khoá}
function RSTSearch(X: TKey): PNode;
var
  b: Integer;
  p: PNode;
begin
  b := z; p := Root; {Bắt đầu với nút gốc, đối với RST thì gốc luôn có sẵn}
  repeat
    b := b - 1; {Xét bit b của X}
    if <Bit b của X là 0> then p := p^.Left  {Gặp 0 rẽ trái}
    else p := p^.Right; {Gặp 1 rẽ phải}
  until (p = nil) or (b = 0);
  RSTSearch := p;
end;
```

Thao tác chèn một giá trị X vào RST được thực hiện như sau: Đầu tiên, ta đứng ở gốc và duyệt dãy bit của X từ trái qua phải (từ bit z - 1 về bit 0), cứ gặp 0 thì rẽ trái, gặp 1 thì rẽ phải. Nếu quá trình rẽ theo một liên kết nil (đi tới nút rỗng) thì lập tức tạo ra một nút mới, và nối vào theo liên kết đó để có đường đi tiếp. Sau khi duyệt hết dãy bit của X, ta sẽ dừng lại ở một nút lá của RST, và công việc cuối cùng là đặt giá trị X vào nút lá đó.

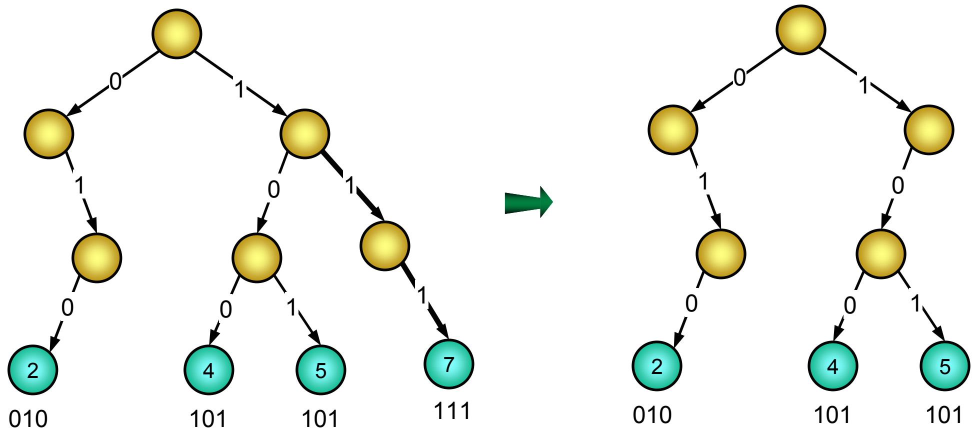
Ví dụ:



Hình 45: Với độ dài dãy bit z = 3, cây tìm kiếm cơ số gồm các khoá 2, 4, 5 và sau khi thêm giá trị 7

```
{Thủ tục chèn khoá X vào cây tìm kiếm cơ số}
procedure RSTInsert(X: TKey);
var
  b: Integer;
  p, q: PNode;
begin
  b := z; p := Root; {Bắt đầu từ nút gốc, đối với RST thì gốc luôn ≠ nil}
  repeat
    b := b - 1; {Xét bit b của X}
    q := p; {Khi p chạy xuống nút con thì q^ luôn giữ vai trò là nút cha của p^}
    if <Bit b của X là 0> then p := p^.Left  {Gặp 0 rẽ trái}
    else p := p^.Right; {Gặp 1 rẽ phải}
    if p = nil then {Không đi được thì  đặt thêm nút để đi tiếp}
      begin
        New(p); {Tạo ra một nút mới và đem p trỏ tới nút đó}
        p^.Left := nil;  p^.Right := nil;
        if <Bit b của X là 0> then q^.Left := p  {Nối p^ vào bên trái q^}
        else q^.Right := p; {Nối p^ vào bên phải q^}
      end;
  until b = 0;
  p^.Info := X; {p^ là nút lá để đặt X vào}
end;
```

Với cây tìm kiếm cơ số, việc xoá một giá trị khoá không phải chỉ là xoá riêng một nút lá mà còn phải xoá toàn bộ nhánh độc đạo đi tới nút đó để tránh lãng phí bộ nhớ (Hình 46).



Hình 46: RST chứa các khoá 2, 4, 5, 7 và RST sau khi loại bỏ giá trị 7

Ta lặp lại quá trình tìm kiếm giá trị khoá X, quá trình này sẽ đi từ gốc xuống lá, tại mỗi bước đi, mỗi khi gặp một nút ngã ba (nút có cả con trái và con phải - nút cấp hai), ta ghi nhận lại ngã ba đó và hướng rẽ. Kết thúc quá trình tìm kiếm ta giữ lại được ngã ba đi qua cuối cùng, từ nút đó tới nút lá chứa X là con đường độc đạo (không có chỗ rẽ), ta tiến hành dỡ bỏ tất cả các nút trên đoạn đường độc đạo khỏi cây tìm kiếm cơ số. Để không bị gặp lỗi khi cây suy biến (không có nút cấp 2) ta coi gốc cũng là nút ngã ba

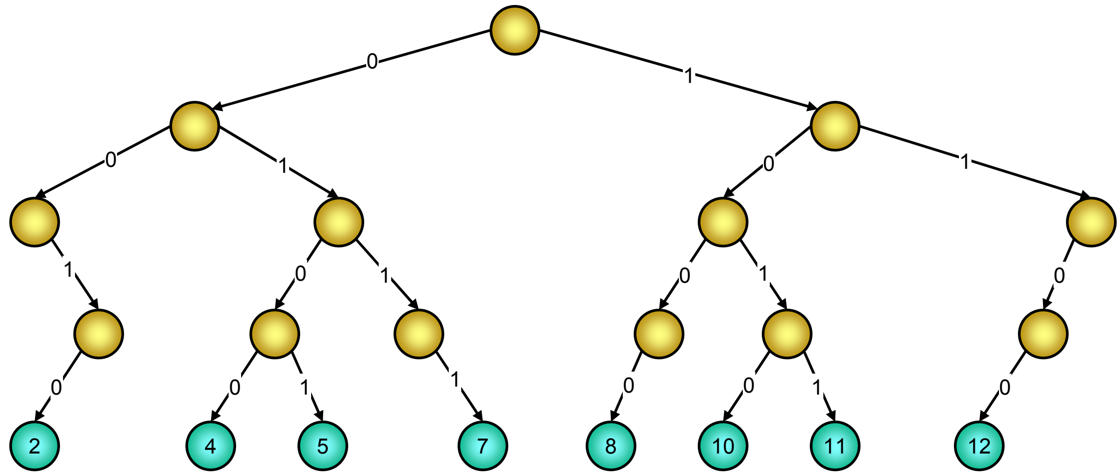
```
{Thủ tục xoá khoá X khỏi cây tìm kiếm cơ số}
procedure RSTDelete(X: TKey);
var
  b: Integer;
  p, q, TurnNode, Child: PNode;
begin
  {Trước hết, tìm kiếm giá trị X xem nó nằm ở nút nào}
  b := z; p := Root;
  repeat
    b := b - 1;
    q := p; {Mỗi lần p chuyển sang nút con, ta luôn đảm bảo cho q^ là nút cha của p^}
    if <Bit b của X là 0> then p := p^.Left
    else p := p^.Right;
    if (b = z - 1) or (q^.Left ≠ nil) and (q^.Right ≠ nil) then {q^ là nút ngã ba}
      begin
        TurnNode := q; Child := p; {Ghi nhận lại q^ và hướng rẽ}
      end;
  until (p = nil) or (b = 0);
  if p = nil then Exit; {X không tồn tại trong cây thì không xoá được}
  {Trước hết, cắt nhánh độc đạo ra khỏi cây}
  if TurnNode^.Left = Child then TurnNode^.Left := nil
  else TurnNode^.Right := nil
  p := Child; {Chuyển sang đoạn đường độc đạo, bắt đầu xoá}
  repeat
    q := p;
    {Lưu ý rằng p^ chỉ có tối đa một nhánh con mà thôi, cho p trỏ sang nhánh con duy nhất nếu có}
    if p^.Left ≠ nil then p := p^.Left
    else p := p^.Right;
    Dispose(q); {Giải phóng bộ nhớ cho nút q^}
  until p = nil;
end;
```

Ta có một nhận xét là: Hình dáng của cây tìm kiếm cơ số không phụ thuộc vào thứ tự chèn các khoá vào mà chỉ phụ thuộc vào giá trị của các khoá chứa trong cây.

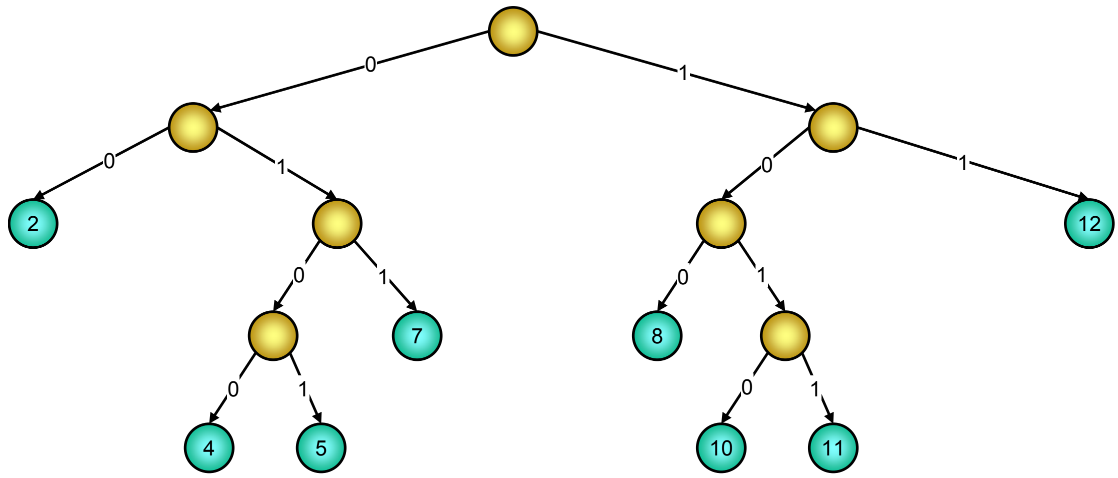
Đối với cây tìm kiếm cơ số, độ phức tạp tính toán cho các thao tác tìm kiếm, chèn, xoá trong trường hợp xấu nhất cũng như trung bình đều là O(z). Do không phải so sánh giá trị khoá dọc đường đi, nó nhanh hơn cây tìm kiếm số học nếu như gặp các khoá cấu trúc lớn. Tốc độ như vậy có thể nói là tốt, nhưng vấn đề bộ nhớ khiến ta phải xem xét: Giá trị chứa trong các nút nhánh của cây tìm kiếm cơ số là vô nghĩa dẫn tới sự lãng phí bộ nhớ.

Một giải pháp cho vấn đề này là: Duy trì hai dạng nút trên cây tìm kiếm cơ số: Dạng nút nhánh chỉ chứa các liên kết trái, phải và dạng nút lá chỉ chứa giá trị khoá. Cài đặt cây này trên một số ngôn ngữ định kiểu quá mạnh đôi khi rất khó.

Giải pháp thứ hai là đặc tả một cây tương tự như RST, nhưng sửa đổi một chút: nếu có nút lá chứa giá trị X được nối với cây bằng một nhánh độc đạo thì cắt bỏ nhánh độc đạo đó, và thay vào chỗ nhánh này chỉ một nút chứa giá trị X. Như vậy các giá trị khoá vẫn chỉ chứa trong các nút lá nhưng các nút lá giờ đây không chỉ nằm trên mức z + 1 mà còn nằm trên những mức khác nữa. Phương pháp này không những tiết kiệm bộ nhớ hơn mà còn làm cho quá trình tìm kiếm nhanh hơn. Giá phải trả cho phương pháp này là thao tác chèn, xoá khá phức tạp. Tên của cấu trúc dữ liệu này là Trie (Trie chứ không phải Tree) tìm kiếm cơ số.



a)



b)

Hình 47: Cây tìm kiếm cơ số a) và Trie tìm kiếm cơ số b)

Tương tự như phương pháp sắp xếp bằng cơ số, phép tìm kiếm bằng cơ số không nhất thiết phải chọn hệ cơ số 2. Ta có thể chọn hệ cơ số lớn hơn để có tốc độ nhanh hơn (kèm theo sự tốn kém bộ nhớ), chỉ lưu ý là cây tìm kiếm số học cũng như cây tìm kiếm cơ số trong trường hợp này không còn là cây nhị phân mà là cây R_phân với R là hệ cơ số được chọn.

Trong các phương pháp tìm kiếm bằng cơ số, thực ra còn một phương pháp tinh tuý và thông minh nhất, nó có cấu trúc gần giống như cây nhưng không có nút dư thừa, và quá trình duyệt bit của khoá tìm kiếm không phải từ trái qua phải mà theo thứ tự của các bit kiểm soát lưu tại mỗi nút đi qua. Phương pháp đó có tên gọi là Practical Algorithm To Retrieve Information Coded In Alphanumeric (PATRICIA) do Morrison đề xuất. Tuy nhiên, việc cài đặt phương pháp này khá phức tạp (đặc biệt là thao tác xoá giá trị khoá), ta có thể tham khảo nội dung của nó trong các tài liệu khác.

9.9. NHỮNG NHẬN XÉT CUỐI CÙNG

Tìm kiếm thường là công việc nhanh hơn sắp xếp nhưng lại được sử dụng nhiều hơn. Trên đây, ta đã trình bày phép tìm kiếm trong một tập hợp để tìm ra bản ghi mang khoá đúng bằng khoá tìm kiếm. Tuy nhiên, người ta có thể yêu cầu tìm bản ghi mang khoá lớn hơn hay nhỏ hơn khoá tìm kiếm, tìm bản ghi mang khoá nhỏ nhất mà lớn hơn khoá tìm kiếm, tìm bản ghi mang khoá lớn nhất mà nhỏ hơn khoá tìm kiếm v.v… Để cài đặt những thuật toán nêu trên cho những trường hợp này cần có một sự mềm dẻo nhất định.

Cũng tương tự như sắp xếp, ta không nên đánh giá giải thuật tìm kiếm này tốt hơn giải thuật tìm kiếm khác. Sử dụng thuật toán tìm kiếm phù hợp với từng yêu cầu cụ thể là kỹ năng của người lập trình, việc cài đặt cây nhị phân tìm kiếm hay cây tìm kiếm cơ số chỉ để tìm kiếm trên vài chục bản ghi chỉ khẳng định được một điều rõ ràng: không biết thế nào là giải thuật và lập trình.

Bài tập

Bài 1

Hãy thử viết một chương trình SearchDemo tương tự như chương trình SortDemo trong bài trước. Đồng thời viết thêm vào chương trình SortDemo ở bài trước thủ tục TreeSort và đánh giá tốc độ thực của nó.

Bài 2

Tìm hiểu các phương pháp tìm kiếm ngoài, cấu trúc của các B_cây

Bài 3

Tìm hiểu các phương pháp tìm kiếm chuỗi, thuật toán BRUTE-FORCE, thuật toán KNUTH-MORRIS-PRATT, thuật toán BOYER-MOORE và thuật toán RABIN-KARP

Tuy gọi là chuyên đề về "Cấu trúc dữ liệu và giải thuật" nhưng thực ra, ta mới chỉ tìm hiểu qua về hai dạng cấu trúc dữ liệu hay gặp là danh sách và cây, cùng với một số thuật toán mà "đâu cũng phải có" là tìm kiếm và sắp xếp. Không một tài liệu nào có thể đề cập tới mọi cấu trúc dữ liệu và giải thuật bởi chúng quá phong phú và liên tục được bổ sung. Những cấu trúc dữ liệu và giải thuật không "phổ thông" lắm như lý thuyết đồ thị, hình học, v.v… sẽ được tách ra và sẽ được nói kỹ hơn trong một chuyên đề khác.

Việc đi sâu nghiên cứu những cấu trúc dữ liệu và giải thuật, dù chỉ là một phần nhỏ hẹp cũng nảy sinh rất nhiều vấn đề hay và khó, như các vấn đề lý thuyết về độ phức tạp tính toán, vấn đề NP_đầy đủ v.v… Đó là công việc của những nhà khoa học máy tính. Nhưng trước khi trở thành một nhà khoa học máy tính thì điều kiện cần là phải **biết lập trình**. Vậy nên khi tìm hiểu bất cứ cấu trúc dữ liệu hay giải thuật nào, nhất thiết ta phải cố gắng cài đặt bằng được. Mọi ý tưởng hay sẽ chỉ là bỏ đi nếu như không biến thành hiệu quả, thực tế là như vậy.

PHẦN 3. QUY HOẠCH ĐỘNG

Các thuật toán đệ quy có ưu điểm dễ cài đặt, tuy nhiên do bản chất của quá trình đệ quy, các chương trình này thường kéo theo những đòi hỏi lớn về không gian bộ nhớ và một khối lượng tính toán khổng lồ.

Quy hoạch động (Dynamic programming) là một kỹ thuật nhằm đơn giản hóa việc tính toán các công thức truy hồi bằng cách lưu trữ toàn bộ hay một phần kết quả tính toán tại mỗi bước với mục đích sử dụng lại. Bản chất của quy hoạch động là thay thế mô hình tính toán "từ trên xuống" (Top-down) bằng mô hình tính toán "từ dưới lên" (Bottom-up).

Từ "programming" ở đây không liên quan gì tới việc lập trình cho máy tính, đó là một thuật ngữ mà các nhà toán học hay dùng để chỉ ra các bước chung trong việc giải quyết một dạng bài toán hay một lớp các vấn đề. Không có một thuật toán tổng quát để giải tất cả các bài toán quy hoạch động.

Mục đích của phần này là cung cấp một cách tiếp cận mới trong việc giải quyết các bài toán tối ưu mang bản chất đệ quy, đồng thời đưa ra các ví dụ để người đọc có thể làm quen và hình thành các kỹ năng trong việc tiếp cận các bài toán quy hoạch động.

§1. CÔNG THỨC TRUY HỒI

1.1. VÍ DỤ

Cho số tự nhiên n ≤ 100. Hãy cho biết có bao nhiêu cách phân tích số n thành tổng của dãy các số nguyên dương, các cách phân tích là hoán vị của nhau chỉ tính là một cách.

Ví dụ: n = 5 có 7 cách phân tích:

```
1. 5 = 1 + 1 + 1 + 1 + 1
2. 5 = 1 + 1 + 1 + 2
3. 5 = 1 + 1 + 3
4. 5 = 1 + 2 + 2
5. 5 = 1 + 4
6. 5 = 2 + 3
7. 5 = 5
```

(Lưu ý: n = 0 vẫn coi là có 1 cách phân tích thành tổng các số nguyên dương (0 là tổng của dãy rỗng))

Để giải bài toán này, trong chuyên mục trước ta đã dùng phương pháp liệt kê tất cả các cách phân tích và đếm số cấu hình. Bây giờ ta thử nghĩ xem, **có cách nào tính ngay ra số lượng các cách phân tích mà không cần phải liệt kê hay không ?**. Bởi vì khi số cách phân tích tương đối lớn, phương pháp liệt kê tỏ ra khá chậm. (n = 100 có 190569292 cách phân tích).

Nhận xét:

Nếu gọi **F[m, v] là số cách phân tích số v thành tổng các số nguyên dương ≤ m**. Khi đó:

Các cách phân tích số v thành tổng các số nguyên dương ≤ m có thể chia làm hai loại:

Loại 1: Không chứa số m trong phép phân tích, khi đó số cách phân tích loại này chính là số cách phân tích số v thành tổng các số nguyên dương < m, tức là số cách phân tích số v thành tổng các số nguyên dương ≤ m - 1 và bằng F[m - 1, v].

Loại 2: Có chứa ít nhất một số m trong phép phân tích. Khi đó nếu trong các cách phân tích loại này ta bỏ đi số m đó thì ta sẽ được các cách phân tích số v - m thành tổng các số nguyên dương ≤ m (Lưu ý: điều này chỉ đúng khi không tính lặp lại các hoán vị của một cách). Có nghĩa là về mặt số lượng, số các cách phân tích loại này bằng F[m, v - m]

Trong trường hợp m > v thì rõ ràng chỉ có các cách phân tích loại 1, còn trong trường hợp m ≤ v thì sẽ có cả các cách phân tích loại 1 và loại 2. Vì thế:

F[m, v] = F[m - 1, v] nếu m > v

F[m, v] = F[m - 1, v] + F[m, v - m] nếu m ≤ v

Ta có công thức xây dựng F[m, v] từ F[m - 1, v] và F[m, v - m]. Công thức này có tên gọi là **công thức truy hồi** đưa việc tính F[m, v] về việc tính các F[m', v'] với dữ liệu nhỏ hơn. Tất nhiên cuối cùng ta sẽ quan tâm đến F[n, n]: Số các cách phân tích n thành tổng các số nguyên dương ≤ n.

Ví dụ với n = 5, bảng F sẽ là:

F (m \ v)	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	3	4	5
4	1	1	2	3	5	6
5	1	1	2	3	5	7

Nhìn vào bảng F, ta thấy rằng F[m, v] được tính bằng tổng của:

Một phần tử ở hàng trên: F[m - 1, v] và một phần tử ở cùng hàng, bên trái: F[m, v - m].

Ví dụ F[5, 5] sẽ được tính bằng F[4, 5] + F[5, 0], hay F[3, 5] sẽ được tính bằng F[2, 5] + F[3, 2]. Chính vì vậy để tính F[m, v] thì F[m - 1, v] và F[m, v - m] phải được tính trước. Suy ra thứ tự hợp lý để tính các phần tử trong bảng F sẽ phải là theo thứ tự từ trên xuống và trên mỗi hàng thì tính theo thứ tự từ trái qua phải.

Điều đó có nghĩa là ban đầu ta phải tính hàng 0 của bảng: F[0, v] = số dãy có các phần tử ≤ 0 mà tổng bằng v, theo quy ước ở đề bài thì F[0, 0] = 1 còn F[0, v] với mọi v > 0 đều là 0.

Vậy giải thuật dựng rất đơn giản: Khởi tạo dòng 0 của bảng F: F[0, 0] = 1 còn F[0, v] với mọi v > 0 đều bằng 0, sau đó dùng công thức truy hồi tính ra tất cả các phần tử của bảng F. Cuối cùng F[n, n] là số cách phân tích cần tìm

P_3_01_1.PAS * Đếm số cách phân tích số n

```
program Analyse1; {Bài toán phân tích số}
const
  max = 100;
var
  F: array[0..max, 0..max] of LongInt;
  n, m, v: Integer;
begin
  Write('n = '); ReadLn(n);
  FillChar(F[0], SizeOf(F[0]), 0); {Khởi tạo dòng 0 của bảng F toàn số 0}
  F[0, 0] := 1; {Duy chỉ có F[0, 0] = 1}
  for m := 1 to n do {Dùng công thức tính các dòng theo thứ tự từ trên xuống dưới}
    for v := 0 to n do {Các phần tử trên một dòng thì tính theo thứ tự từ trái qua phải}
      if v < m then F[m, v] := F[m - 1, v]
      else F[m, v] := F[m - 1, v] + F[m, v - m];
  WriteLn(F[n, n], ' Analyses'); {Cuối cùng F[n, n] là số cách phân tích}
end.
```

1.2. CẢI TIẾN THỨ NHẤT

Cách làm trên có thể tóm tắt lại như sau: Khởi tạo dòng 0 của bảng, sau đó dùng dòng 0 tính dòng 1, dùng dòng 1 tính dòng 2 v.v… tới khi tính được hết dòng n. Có thể nhận thấy rằng khi đã tính xong dòng thứ k thì việc lưu trữ các dòng từ dòng 0 tới dòng k - 1 là không cần thiết bởi vì việc tính dòng k + 1 chỉ phụ thuộc các giá trị lưu trữ trên dòng k. Vậy ta có thể dùng hai mảng một chiều: Mảng Current lưu dòng hiện thời đang xét của bảng và mảng Next

lưu dòng kế tiếp, đầu tiên mảng Current được gán các giá trị tương ứng trên dòng 0. Sau đó dùng mảng Current tính mảng Next, mảng Next sau khi tính sẽ mang các giá trị tương ứng trên dòng 1. Rồi lại gán mảng Current := Next và tiếp tục dùng mảng Current tính mảng Next, mảng Next sẽ gồm các giá trị tương ứng trên dòng 2 v.v… Vậy ta có cài đặt cải tiến sau:

P_3_01_2.PAS * Đếm số cách phân tích số n

```
program Analyse2;
const
  max = 100;
var
  Current, Next: array[0..max] of LongInt;
  n, m, v: Integer;
begin
  Write('n = '); ReadLn(n);
  FillChar(Current, SizeOf(Current), 0);
  Current[0] := 1; {Khởi tạo mảng Current tương ứng với dòng 0 của bảng F}
  for m := 1 to n do
    begin {Dùng dòng hiện thời Current tính dòng kế tiếp Next ⇔ Dùng dòng m - 1 tính dòng m của bảng F}
      for v := 0 to n do
        if v < m then Next[v] := Current[v]
        else Next[v] := Current[v] + Next[v - m];
      Current := Next; {Gán Current := Next tức là Current bây giờ lại lưu các phần tử trên dòng m của bảng F}
    end;
  WriteLn(Current[n], ' Analyses');
end.
```

Cách làm trên đã tiết kiệm được khá nhiều không gian lưu trữ, nhưng nó hơi chậm hơn phương pháp đầu tiên vì phép gán mảng (Current := Next). Có thể cải tiến thêm cách làm này như sau:

P_3_01_3.PAS * Đếm số cách phân tích số n

```
program Analyse3;
const
  max = 100;
var
  B: array[1..2, 0..max] of LongInt;{Bảng B chỉ gồm 2 dòng thay cho 2 dòng liên tiếp của bảng phương án}
  n, m, v, x, y: Integer;
begin
  Write('n = '); ReadLn(n);
  {Trước hết, dòng 1 của bảng B tương ứng với dòng 0 của bảng phương án F, được điền cơ sở quy hoạch động}
  FillChar(B[1], SizeOf(B[1]), 0);
  B[1][0] := 1;
  x := 1; {Dòng B[x] đóng vai trò là dòng hiện thời trong bảng phương án}
  y := 2; {Dòng B[y] đóng vai trò là dòng kế tiếp trong bảng phương án}
  for m := 1 to n do
    begin
   {Dùng dòng x tính dòng y ⇔ Dùng dòng hiện thời trong bảng phương án để tính dòng kế tiếp}
      for v := 0 to n do
        if v < m then B[y][v] := B[x][v]
        else B[y][v] := B[x][v] + B[y][v - m];
      x := 3 - x; y := 3 - y; {Đảo giá trị x và y, tính xoay lại}
    end;
  WriteLn(B[x][n], ' Analyses');
end.
```

1.3. CẢI TIẾN THỨ HAI

Ta vẫn còn cách tốt hơn nữa, tại mỗi bước, ta chỉ cần lưu lại một dòng của bảng F bằng một mảng 1 chiều, sau đó dùng mảng đó tính lại chính nó để sau khi tính, mảng một chiều sẽ lưu các giá trị của bảng F trên dòng kế tiếp.

P_3_01_4.PAS * Đếm số cách phân tích số n

```
program Analyse4;
const
  max = 100;
var
  L: array[0..max] of LongInt; {Chỉ cần lưu 1 dòng}
  n, m, v: Integer;
begin
  Write('n = '); ReadLn(n);
  FillChar(L, SizeOf(L), 0);
  L[0] := 1; {Khởi tạo mảng 1 chiều L lưu dòng 0 của bảng}
  for m := 1 to n do    {Dùng L tính lại chính nó}
    for v := m to n do
      L[v] := L[v] + L[v - m];
  WriteLn(L[n], ' Analyses');
end.
```

1.4. CÀI ĐẶT ĐỆ QUY

Xem lại công thức truy hồi tính F[m, v] = F[m - 1, v] + F[m, v - m], ta nhận thấy rằng để tính F[m, v] ta phải biết được chính xác F[m - 1, v] và F[m, v - m]. Như vậy việc xác định thứ tự tính các phần tử trong bảng F (phần tử nào tính trước, phần tử nào tính sau) là quan trọng. Tuy nhiên ta có thể tính dựa trên một hàm đệ quy mà không cần phải quan tâm tới thứ tự tính toán. Việc viết một hàm đệ quy tính công thức truy hồi khá đơn giản, như ví dụ này ta có thể viết:

P_3_01_5.PAS * Đếm số cách phân tích số n dùng đệ quy

```
program Analyse5;
var
  n: Integer;

function GetF(m, v: Integer): LongInt;
begin
  if m = 0 then   {Phần neo của hàm đệ quy}
    if v = 0 then GetF := 1
    else GetF := 0
  else {Phần đệ quy}
    if m > v then GetF := GetF(m - 1, v)
    else GetF := GetF(m - 1, v) + GetF(m, v - m);
end;

begin
  Write('n = '); ReadLn(n);
  WriteLn(GetF(n, n), ' Analyses');
end.
```

Phương pháp cài đặt này tỏ ra khá chậm vì phải gọi nhiều lần mỗi hàm GetF(m, v) (bài sau sẽ giải thích rõ hơn điều này). Ta có thể cải tiến bằng cách kết hợp với một mảng hai chiều F. Ban đầu các phần tử của F được coi là "chưa biết" (bằng cách gán một giá trị đặc biệt). Hàm GetF(m, v) khi được gọi trước hết sẽ tra cứu tới F[m, v], nếu F[m, v] chưa biết thì hàm

GetF(m, v) sẽ gọi đệ quy để tính giá trị của F[m, v] rồi dùng giá trị này gán cho kết quả hàm, còn nếu F[m, v] đã biết thì hàm này chỉ việc gán kết quả hàm là F[m, v] mà không cần gọi đệ quy để tính toán nữa.

P_3_01_6.PAS * Đếm số cách phân tích số n dùng đệ quy

```
program Analyse6;
const
  max = 100;
var
  n: Integer;
  F: array[0..max, 0..max] of LongInt;

function GetF(m, v: Integer): LongInt;
begin
  if F[m, v] = -1 then {Nếu F[m, v] chưa biết thì đi tính F[m, v]}
    begin
      if m = 0 then   {Phần neo của hàm đệ quy}
        if v = 0 then F[m, v] := 1
        else F[m, v] := 0
      else {Phần đệ quy}
        if m > v then F[m, v] := GetF(m - 1, v)
        else F[m, v] := GetF(m - 1, v) + GetF(m, v - m);
    end;
  GetF := F[m, v]; {Gán kết quả hàm bằng F[m, v]}
end;

begin
  Write('n = '); ReadLn(n);
  FillChar(f, SizeOf(f), $FF); {Khởi tạo mảng F bằng giá trị -1}
  WriteLn(GetF(n, n), ' Analyses');
end.
```

Việc sử dụng phương pháp đệ quy để giải công thức truy hồi là một kỹ thuật đáng lưu ý, vì khi gặp một công thức truy hồi phức tạp, khó xác định thứ tự tính toán thì phương pháp này tỏ ra rất hiệu quả, hơn thế nữa nó làm rõ hơn bản chất đệ quy của công thức truy hồi.

§2. PHƯƠNG PHÁP QUY HOẠCH ĐỘNG

2.1. BÀI TOÁN QUY HOẠCH

Bài toán quy hoạch là **bài toán tối ưu:** gồm có một hàm f gọi là hàm mục tiêu hay hàm đánh giá; các hàm g1, g2, …, gn cho giá trị logic gọi là hàm ràng buộc. Yêu cầu của bài toán là tìm một cấu hình x thoả mãn tất cả các ràng buộc g1, g2, …gn: gi(x) = TRUE (∀i: 1 ≤ i ≤ n) và x là tốt nhất, theo nghĩa không tồn tại một cấu hình y nào khác thoả mãn các hàm ràng buộc mà f(y) tốt hơn f(x).

Ví dụ:

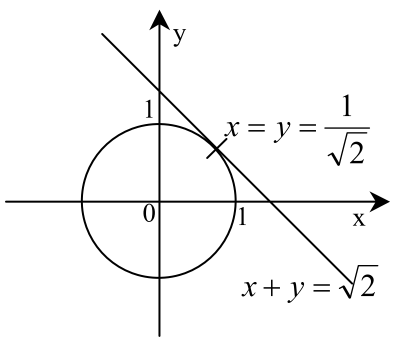
Tìm (x, y) để

Hàm mục tiêu : x + y → max

Hàm ràng buộc : $x^2 + y^2 \le 1$.

Xét trong mặt phẳng toạ độ, những cặp (x, y) thoả mãn $x^2 + y^2 \le 1$ là tọa độ của những điểm nằm trong hình tròn có tâm O là gốc toạ độ, bán kính 1. Vậy nghiệm của bài toán bắt buộc nằm trong hình tròn đó.

Những đường thẳng có phương trình: x + y = C (C là một hằng số) là đường thẳng vuông góc với đường phân giác góc phần tư thứ nhất. Ta phải tìm số C lớn nhất mà đường thẳng x + y = C vẫn có điểm chúng với đường tròn (O, 1). Đường thẳng đó là một tiếp tuyến của đường tròn: $x + y = \sqrt{2}$. Tiếp điểm $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$ tương ứng với nghiệm tối ưu của bài toán đã cho.



Các dạng bài toán quy hoạch rất phong phú và đa dạng, ứng dụng nhiều trong thực tế, nhưng cũng cần biết rằng, đa số các bài toán quy hoạch là không giải được, hoặc chưa giải được. Cho đến nay, người ta mới chỉ có thuật toán đơn hình giải bài toán quy hoạch tuyến tính lồi, và một vài thuật toán khác áp dụng cho các lớp bài toán cụ thể.

2.2. PHƯƠNG PHÁP QUY HOẠCH ĐỘNG

Phương pháp quy hoạch động dùng để giải bài toán tối ưu có bản chất đệ quy, tức là việc tìm phương án tối ưu cho bài toán đó có thể đưa về tìm phương án tối ưu của một số hữu hạn các

bài toán con. Đối với nhiều thuật toán đệ quy chúng ta đã tìm hiểu, nguyên lý chia để trị (divide and conquer) thường đóng vai trò chủ đạo trong việc thiết kế thuật toán. Để giải quyết một bài toán lớn, ta chia nó làm nhiều bài toán con cùng dạng với nó để có thể giải quyết độc lập. Trong phương pháp quy hoạch động, nguyên lý này càng được thể hiện rõ: Khi không biết cần phải giải quyết những bài toán con nào, ta sẽ đi **giải quyết tất cả các bài toán con** và **lưu trữ những lời giải hay đáp số của chúng** với mục đích **sử dụng lại** theo một sự phối hợp nào đó để giải quyết những bài toán tổng quát hơn. Đó chính là điểm khác nhau giữa Quy hoạch động và phép phân giải đệ quy và cũng là nội dung phương pháp quy hoạch động:

Phép phân giải đệ quy **bắt đầu từ bài toán lớn** phân rã thành nhiều bài toán con và đi giải từng bài toán con đó. Việc giải từng bài toán con lại đưa về phép phân rã tiếp thành nhiều bài toán nhỏ hơn và lại đi giải tiếp bài toán nhỏ hơn đó bất kể nó đã được giải hay chưa.

Quy hoạch động **bắt đầu từ việc giải tất cả các bài toán nhỏ nhất** (bài toán cơ sở) để từ đó từng bước giải quyết những bài toán lớn hơn, cho tới khi giải được bài toán lớn nhất (bài toán ban đầu).

Ta xét một ví dụ đơn giản:

Ví dụ: Dãy Fibonacci là dãy số nguyên dương được định nghĩa như sau:

$F_1 = F_2 = 1;$

$\forall i: 3 \le i: F_i = F_{i-1} + F_{i-2}$

Hãy tính F_6

Xét hai cách cài đặt chương trình:

```
Cách 1
program Fibo1;

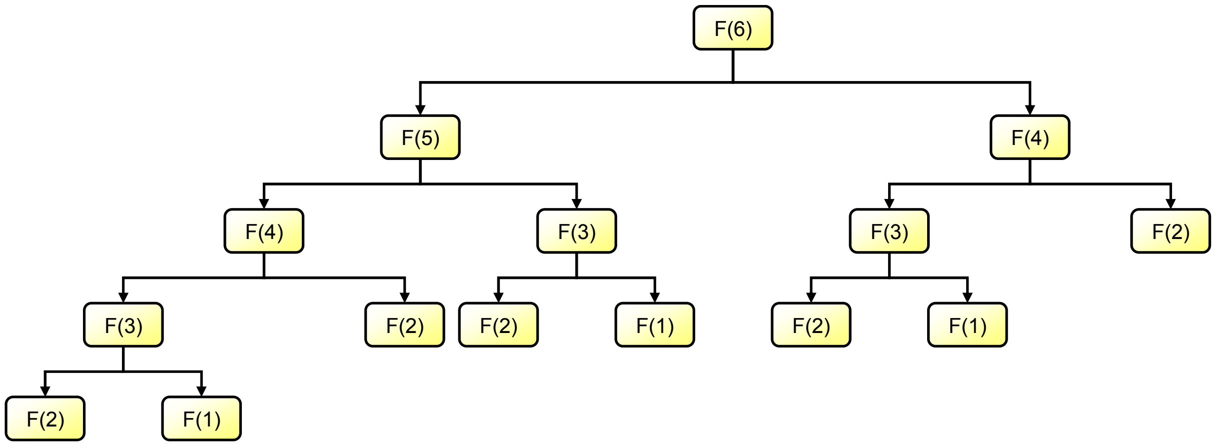
function F(i: Integer): Integer;
begin
  if i < 3 then F := 1
  else F := F(i - 1) + F(i - 2);
end;

begin
  WriteLn(F(6));
end.
```

```
Cách 2
program Fibo2;
var
  F: array[1..6] of Integer;
  i: Integer;

begin
  F[1] := 1; F[2] := 1;
  for i := 3 to 6 do
    F[i] := F[i - 1] + F[i - 2];
  WriteLn(F[6]);
end.
```

Trong cách 1, ta viết một hàm đệ quy F(i) để tính số Fibonacci thứ i. Chương trình chính gọi F(6), nó sẽ gọi tiếp F(5) và F(4) để tính … Quá trình tính toán có thể vẽ như cây dưới đây. Ta nhận thấy để tính F(6) nó phải tính 1 lần F(5), hai lần F(4), ba lần F(3), năm lần F(2), ba lần F(1).



Hình 48: Hàm đệ quy tính số Fibonacci

Cách 2 thì không như vậy. Trước hết nó tính sẵn F[1] và F[2], từ đó tính tiếp F[3], lại tính tiếp được F[4], F[5], F[6]. Đảm bảo rằng mỗi giá trị Fibonacci chỉ phải tính 1 lần.

(Cách 2 còn có thể cải tiến thêm nữa, chỉ cần dùng 3 giá trị tính lại lẫn nhau)

Trước khi áp dụng phương pháp quy hoạch động ta phải xét xem phương pháp đó có thoả mãn những yêu cầu dưới đây hay không:

Bài toán lớn phải phân rã được thành nhiều bài toán con, mà sự phối hợp lời giải của các bài toán con đó cho ta lời giải của bài toán lớn.

Vì quy hoạch động là đi **giải tất cả** các bài toán con, nên nếu không đủ không gian vật lý lưu trữ lời giải (bộ nhớ, đĩa…) để phối hợp chúng thì phương pháp quy hoạch động cũng không thể thực hiện được.

Quá trình từ bài toán cơ sở tìm ra lời giải bài toán ban đầu phải qua hữu hạn bước.

Các khái niệm:

Bài toán giải theo phương pháp quy hoạch động gọi là **bài toán quy hoạch động**

Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn gọi là **công thức truy hồi** (hay phương trình truy toán) của quy hoạch động

Tập các bài toán nhỏ nhất có ngay lời giải để từ đó giải quyết các bài toán lớn hơn gọi là **cơ sở quy hoạch động**

Không gian lưu trữ lời giải các bài toán con để tìm cách phối hợp chúng gọi là **bảng phương án của quy hoạch động**

Các bước cài đặt một chương trình sử dụng quy hoạch động: (nhớ kỹ)

Giải tất cả các bài toán cơ sở (thông thường rất dễ), lưu các lời giải vào bảng phương án.

Dùng công thức truy hồi phối hợp những lời giải của những bài toán nhỏ đã lưu trong bảng phương án để tìm lời giải của những bài toán lớn hơn và lưu chúng vào bảng phương án. Cho tới khi bài toán ban đầu tìm được lời giải.

Dựa vào bảng phương án, truy vết tìm ra nghiệm tối ưu.

Cho đến nay, vẫn chưa có một định lý nào cho biết một cách chính xác những bài toán nào có thể giải quyết hiệu quả bằng quy hoạch động. Tuy nhiên để biết được bài toán có thể giải bằng quy hoạch động hay không, ta có thể tự đặt câu hỏi: **"Một nghiệm tối ưu của bài toán lớn có phải là sự phối hợp các nghiệm tối ưu của các bài toán con hay không ?"** và **"Liệu có thể nào lưu trữ được nghiệm các bài toán con dưới một hình thức nào đó để phối hợp tìm được nghiệm bài toán lớn"**

§3. MỘT SỐ BÀI TOÁN QUY HOẠCH ĐỘNG

3.1. DÃY CON ĐƠN ĐIỆU TĂNG DÀI NHẤT

Cho dãy số nguyên A = $a_1, a_2, \ldots, a_n$. ($n \le 5000$, $-10000 \le a_i \le 10000$). Một dãy con của A là một cách chọn ra trong A một số phần tử giữ nguyên thứ tự. Như vậy A có 2^n dãy con.

Yêu cầu: Tìm dãy con đơn điệu tăng của A có độ dài lớn nhất.

Ví dụ: A = (1, 2, 3, 4, 9, 10, 5, 6, 7). Dãy con đơn điệu tăng dài nhất là: (1, 2, 3, 4, 5, 6, 7).

Input: file văn bản INCSEQ.INP

- Dòng 1: Chứa số n
- Dòng 2: Chứa n số $a_1, a_2, \ldots, a_n$ cách nhau ít nhất một dấu cách

Output: file văn bản INCSEQ.OUT

- Dòng 1: Ghi độ dài dãy con tìm được
- Các dòng tiếp: ghi dãy con tìm được và chỉ số những phần tử được chọn vào dãy con đó.

INCSEQ.INP	INCSEQ.OUT
11 1 2 3 8 9 4 5 6 20 9 10	8 a[1] = 1 a[2] = 2 a[3] = 3 a[6] = 4 a[7] = 5 a[8] = 6 a[10] = 9 a[11] = 10

Cách giải:

Bổ sung vào A hai phần tử: $a_0 = -\infty$ và $a_{n+1} = +\infty$. ***Khi đó dãy con đơn điệu tăng dài nhất chắc chắn sẽ bắt đầu từ a_0 và kết thúc ở a_{n+1}.***

Với $\forall$ i: $0 \le i \le n + 1$. Ta sẽ tính L[i] = độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại a_i.

3.1.1. Cơ sở quy hoạch động (bài toán nhỏ nhất):

L[n + 1] = Độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại $a_{n+1} = +\infty$. Dãy con này chỉ gồm mỗi một phần tử ($+\infty$) nên L[n + 1] = 1.

3.1.2. Công thức truy hồi:

Giả sử với i chạy từ n về 0, ta cần tính L[i]: độ dài dãy con tăng dài nhất bắt đầu tại a_i. L[i] được tính trong điều kiện L[i + 1], L[i + 2], …, L[n + 1] đã biết:

Dãy con đơn điệu tăng dài nhất bắt đầu từ a_i sẽ được thành lập bằng cách lấy a_i ghép vào đầu một trong số những dãy con đơn điệu tăng dài nhất bắt đầu tại vị trí a_j đứng sau a_i. Ta sẽ chọn

dãy nào để ghép a_i vào đầu? Tất nhiên là chỉ được ghép a_i vào đầu những dãy con bắt đầu tại a_j nào đó lớn hơn a_i (để đảm bảo tính tăng) và dĩ nhiên ta sẽ chọn dãy dài nhất để ghép a_i vào đầu (để đảm bảo tính dài nhất). Vậy L[i] được tính như sau: **Xét tất cả các chỉ số j trong khoảng từ i + 1 đến n + 1 mà $a_j > a_i$, chọn ra chỉ số jmax có L[jmax] lớn nhất. Đặt L[i] := L[jmax] + 1.**

3.1.3. Truy vết

Tại bước xây dựng dãy L, mỗi khi gán L[i] := L[jmax] + 1, ta đặt T[i] = jmax. Để lưu lại rằng: Dãy con dài nhất bắt đầu tại a_i sẽ có phần tử thứ hai kế tiếp là a_{jmax}.

Sau khi tính xong hay dãy L và T, ta bắt đầu từ 0. T[0] là phần tử đầu tiên được chọn,

T[T[0]] là phần tử thứ hai được chọn,

T[T[T[0]]] là phần tử thứ ba được chọn …Quá trình truy vết có thể diễn tả như sau:

```
i := T[0];
while i <> n + 1 do {Chừng nào chưa duyệt đến số an+1=+∞ ở cuối}
  begin
    <Thông báo chọn ai>
    i := T[i];
  end;
```

Ví dụ: với A = (5, 2, 3, 4, 9, 10, 5, 6, 7, 8). Hai dãy L và T sau khi tính sẽ là:

Calculating ←

i	0	1	2	3	4	5	6	7	8	9	10	11
a_i	$-\infty$	5	2	3	4	9	10	5	6	7	8	$+\infty$
L[i]	9	5	8	7	6	3	2	5	4	3	2	1
T[i]	2	8	3	4	7	6	11	8	9	10	11	

Tracing →

Hình 49: Tính toán và truy vết

P_3_03_1.PAS * Tìm dãy con đơn điệu tăng dài nhất

```
program LongestSubSequence;
const
  InputFile  = 'INCSEQ.INP';
  OutputFile = 'INCSEQ.OUT';
  max = 5000;
var
  a, L, T: array[0..max + 1] of Integer;
  n: Word;

procedure Enter;
var
  i: Word;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n);
  for i := 1 to n do Read(f, a[i]);
  Close(f);
end;
```

```
procedure Optimize; {Quy hoạch động}
var
  i, j, jmax: Word;
begin
  a[0] := -32768; a[n + 1] := 32767; {Thêm hai phần tử canh hai đầu dãy a}
  L[n + 1] := 1; {Điền cơ sở quy hoach động vào bảng phương án}
  for i := n downto 0 do {Tính bảng phương án}
    begin
   {Chọn trong các chỉ số j đứng sau i thoả mãn aj > ai ra chỉ số jmax có L[jmax] lớn nhất}
      jmax := n + 1;
      for j := i + 1 to n + 1 do
        if (a[j] > a[i]) and (L[j] > L[jmax]) then jmax := j;
      L[i] := L[jmax] + 1; {Lưu độ dài dãy con tăng dài nhất bắt đầu tại ai}
      T[i] := jmax; {Lưu vết: phần tử đứng liền sau ai trong dãy con tăng dài nhất đó là ajmax}
    end;
end;

procedure Result;
var
  f: Text;
  i: Integer;
begin
  Assign(f, OutputFile); Rewrite(f);
  WriteLn(f, L[0] - 2); {Chiều dài dãy con tăng dài nhất}
  i := T[0]; {Bắt đầu truy vết tìm nghiệm}
  while i <> n + 1 do
    begin
      WriteLn(f, 'a[', i, '] = ', a[i]);
      i := T[i];
    end;
  Close(f);
end;

begin
  Enter;
  Optimize;
  Result;
end.
```

Nhận xét: Công thức truy hồi tính các L[.] có thể tóm tắt là:

$$\begin{cases} L[n+1] := 0 \\ L[i] := \max\limits_{\substack{i<j\le n+1 \\ a_i<a_j}} L[j]+1 \; (\forall i = \overline{0,n}) \end{cases}$$

và để tính hết các L[.], ta phải mất một đoạn chương trình với độ phức tạp tính toán là $O(n^2)$. Ta có thể cải tiến cách cài đặt để được một đoạn chương trình với độ phức tạp tính toán là O(nlogn) bằng kỹ thuật sau:

Với mỗi số k, ta gọi StartOf[k] là chỉ số x của phần tử a[x] thoả mãn: dãy đơn điệu tăng dài nhất bắt đầu từ a[x] có độ dài k. Nếu có nhiều phần tử a[.] cùng thoả mãn điều kiện này thì ta chọn phần tử a[x] là phần tử lớn nhất trong số những phần tử đó. Việc tính các giá trị StartOf[.] được thực hiện đồng thời với việc tính các giá trị L[.] bằng phương pháp sau:

```
L[n + 1] := 1;
StartOf[1] := n  + 1;
m := 1; {m là độ dài dãy con đơn điệu tăng dài nhất của dãy ai, ai+1, …, an+1 (ở bước khởi tạo này i = n + 1)}
for i := n downto 0 do
  begin
```

```
    <Tính L[i]; đặt k := L[i]>;
    if k > m then {Nếu dãy con tăng dài nhất bắt đầu tại a[i] có độ dài > m}
      begin
        m := k; {Cập nhật lại m}
        StartOf[k] := i; {Gán giá trị cho StartOf[m]}
      end
    else
      if a[i] > a[StartOf[k]] then {Nếu có nhiều dãy đơn điệu tăng dài nhất độ dài k thì}
        StartOf[k] := i; {chỉ ghi nhận lại dãy có phần tử bắt đầu lớn nhất}
  end;
```

Khi bắt đầu vào một lần lặp với một giá trị i, ta đã biết được:

m: Độ dài dãy con đơn điệu tăng dài nhất của dãy $a_{i+1}, a_{i+2}, \dots, a_{n+1}$

StartOf[k] ($1 \le k \le m$): Phần tử $a_{StartOf[k]}$ là phần tử lớn nhất trong số các phần tử $a_{i+1}, a_{i+2}, \dots, a_{n+1}$ thoả mãn: Dãy con đơn điệu tăng dài nhất bắt đầu từ $a_{StartOf[k]}$ có độ dài k. Do thứ tự tính toán được áp đặt như trong sơ đồ trên, ta dễ dàng nhận thấy rằng: $a_{StartOf[k]} < a_{StartOf[k-1]} < \dots < a_{StartOf[1]}$.

Điều kiện để có dãy con đơn điệu tăng độ dài p+1 bắt đầu tại a_i chính là $a_{StartOf[p]} > a_i$ (vì theo thứ tự tính toán thì khi bắt đầu một lần lặp với giá trị i, $a_{StartOf[p]}$ luôn đứng sau a_i). Mặt khác nếu đem a_i ghép vào đầu dãy con đơn điệu tăng dài nhất bắt đầu tại $a_{StartOf[p]}$ mà thu được dãy tăng thì đem a_i ghép vào đầu dãy con đơn điệu tăng dài nhất bắt đầu tại $a_{StartOf[p-1]}$ ta cũng thu được dãy tăng. Vậy để tính L[i], ta có thể tìm số p lớn nhất thoả mãn $a_{StartOf[p]} > a_i$ bằng **thuật toán tìm kiếm nhị phân** rồi đặt L[i] := p + 1 (và sau đó T[i] := StartOf[p], tất nhiên)

P_3_03_2.PAS * Cải tiến thuật toán tìm dãy con đơn điệu tăng dài nhất

```
program LongestSubSequence;
const
  InputFile  = 'INCSEQ.INP';
  OutputFile = 'INCSEQ.OUT';
const
  max = 5000;
var
  a, L, T, StartOf: array[0..max + 1] of Integer;
  n, m: Integer;

procedure Enter;
var
  i: Word;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n);
  for i := 1 to n do Read(f, a[i]);
  Close(f);
end;

procedure Init;
begin
  a[0] := -32768;
  a[n + 1] := 32767;
  m := 1;
  L[n + 1] := 1;
  StartOf[1] := n + 1;
end;
```

```
{Hàm Find, tìm vị trí j mà nếu đem ai ghép vào đầu dãy con đơn điệu tăng dài nhất bắt đầu từ aj sẽ được dãy đơn
điệu tăng dài nhất bắt đầu tại ai}
function Find(i: Integer): Integer;
var
  inf, sup, median, j: Integer;
begin
  inf := 1; sup := m + 1;
  repeat  {Thuật toán tìm kiếm nhị phân}
    median := (inf + sup) div 2;
    j := StartOf[median];
    if a[j] > a[i] then inf := median {Luôn để aStartOf[inf] > ai ≥ aStartOf[sup]}
    else sup := median;
  until inf + 1 = sup;
  Find := StartOf[inf];
end;

procedure Optimize;
var
  i, j, k: Integer;
begin
  for i := n downto 0 do
    begin
      j := Find(i);
      k := L[j] + 1;
      if k > m then
        begin
          m := k;
          StartOf[k] := i;
        end
      else
        if a[StartOf[k]] < a[i] then
          StartOf[k] := i;
      L[i] := k;
      T[i] := j;
    end;
end;

procedure Result;
var
  f: Text;
  i: Integer;
begin
  Assign(f, OutputFile); Rewrite(f);
  WriteLn(f, m - 2);
  i := T[0];
  while i <> n + 1 do
    begin
      WriteLn(f, 'a[', i, '] = ', a[i]);
      i := T[i];
    end;
  Close(f);
end;

begin
  Enter;
  Init;
  Optimize;
  Result;
end.
```

Dễ thấy chi phí thời gian thực hiện giải thuật này cấp O(nlogn), đây là một ví dụ điển hình cho thấy rằng một công thức truy hồi có thể có nhiều phương pháp tính.

3.2. BÀI TOÁN CÁI TÚI

Trong siêu thị có n gói hàng (n ≤ 100), gói hàng thứ i có trọng lượng là $W_i \le 100$ và trị giá $V_i \le 100$. Một tên trộm đột nhập vào siêu thị, tên trộm mang theo một cái túi có thể mang được tối đa trọng lượng M (M ≤ 100). Hỏi tên trộm sẽ lấy đi những gói hàng nào để được tổng giá trị lớn nhất.

Input: file văn bản BAG.INP

- Dòng 1: Chứa hai số n, M cách nhau ít nhất một dấu cách
- n dòng tiếp theo, dòng thứ i chứa hai số nguyên dương W_i, V_i cách nhau ít nhất một dấu cách

Output: file văn bản BAG.OUT

- Dòng 1: Ghi giá trị lớn nhất tên trộm có thể lấy
- Dòng 2: Ghi chỉ số những gói bị lấy

BAG.INP	BAG.OUT
5 11 3 3 4 4 5 4 9 10 4 4	11 5 2 1

Cách giải:

Nếu gọi F**[i, j] là giá trị lớn nhất có thể có** bằng cách chọn trong các gói {1, 2, …, i} với giới hạn trọng lượng j. Thì giá trị lớn nhất khi được chọn trong số n gói với giới hạn trọng lượng M chính là F[n, M].

3.2.1. Công thức truy hồi tính F[i, j].

Với giới hạn trọng lượng j, việc chọn tối ưu trong số các gói {1, 2, …,i - 1, i} để có giá trị lớn nhất sẽ có hai khả năng:

Nếu không chọn gói thứ i thì F[i, j] là giá trị lớn nhất có thể bằng cách chọn trong số các gói {1, 2, …, i - 1} với giới hạn trọng lượng là j. Tức là

$$F[i, j] = F[i - 1, j]$$

Nếu có chọn gói thứ i (tất nhiên chỉ xét tới trường hợp này khi mà $W_i \le j$) thì F[i, j] bằng giá trị gói thứ i là V_i cộng với giá trị lớn nhất có thể có được bằng cách chọn trong số các gói {1, 2, …, i - 1} với giới hạn trọng lượng $j - W_i$. Tức là về mặt giá trị thu được:

$$F[i, j] = V_i + F[i - 1, j - W_i]$$

Vì theo cách xây dựng F[i, j] là giá trị lớn nhất có thể, nên F[i, j] sẽ là max trong 2 giá trị thu được ở trên.

3.2.2. Cơ sở quy hoạch động:

Dễ thấy F[0, j] = giá trị lớn nhất có thể bằng cách chọn trong số 0 gói = 0.

3.2.3. Tính bảng phương án:

Bảng phương án F gồm n + 1 dòng, M + 1 cột, trước tiên được điền cơ sở quy hoạch động: Dòng 0 gồm toàn số 0. Sử dụng công thức truy hồi, dùng dòng 0 tính dòng 1, dùng dòng 1 tính dòng 2, v.v… đến khi tính hết dòng n.

F	0	1	2	......	M
0	0	0	0	...0...	0
1					
2					
...	...	...	...	...	...
n					

3.2.4. Truy vết:

Tính xong bảng phương án thì ta quan tâm đến F[n, M] đó chính là giá trị lớn nhất thu được khi chọn trong cả n gói với giới hạn trọng lượng M. Nếu F[n, M] = F[n - 1, M] thì tức là không chọn gói thứ n, ta truy tiếp F[n - 1, M]. Còn nếu F[n, M] ≠ F[n - 1, M] thì ta thông báo rằng phép chọn tối ưu có chọn gói thứ n và truy tiếp F[n - 1, M - W_n]. Cứ tiếp tục cho tới khi truy lên tới hàng 0 của bảng phương án.

P_3_03_3.PAS * Bài toán cái túi

```
program The_Bag;
const
  InputFile  = 'BAG.INP';
  OutputFile = 'BAG.OUT';
  max = 100;
var
  W, V: Array[1..max] of Integer;
  F: array[0..max, 0..max] of Integer;
  n, M: Integer;

procedure Enter;
var
  i: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, M);
  for i := 1 to n do ReadLn(fi, W[i], V[i]);
  Close(fi);
end;

procedure Optimize; {Tính bảng phương án bằng công thức truy hồi}
var
  i, j: Integer;
begin
  FillChar(F[0], SizeOf(F[0]), 0); {Điền cơ sở quy hoạch động}
```

```
  for i := 1 to n do
    for j := 0 to M do
      begin {Tính F[i, j]}
        F[i, j] := F[i - 1, j]; {Giả sử không chọn gói thứ i thì F[i, j] = F[i - 1, j]}
     {Sau đó đánh giá: nếu chọn gói thứ i sẽ được lợi hơn thì đặt lại F[i, j]}
        if (j >= W[i]) and
              (F[i, j] < F[i - 1, j - W[i]] + V[i]) then
                      F[i, j] := F[i - 1, j - W[i]] + V[i];
      end;
end;

procedure Trace; {Truy vết tìm nghiệm tối ưu}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, F[n, M]); {In ra giá trị lớn nhất có thể kiếm được}
  while n <> 0 do {Truy vết trên bảng phương án từ hàng n lên hàng 0}
    begin
      if F[n, M] <> F[n - 1, M] then {Nếu có chọn gói thứ n}
        begin
          Write(fo, n, ' ');
          M := M - W[n]; {Đã chọn gói thứ n rồi thì chỉ có thể mang thêm được trọng lượng M - Wn nữa thôi}
        end;
      Dec(n);
    end;
  Close(fo);
end;

begin
  Enter;
  Optimize;
  Trace;
end.
```

3.3. BIẾN ĐỔI XÂU

Cho xâu ký tự X, xét 3 phép biến đổi:

a) Insert(i, C): i là số, C là ký tự: Phép Insert chèn ký tự C vào sau vị trí i của xâu X.

b) Replace(i, C): i là số, C là ký tự: Phép Replace thay ký tự tại vị trí i của xâu X bởi ký tự C.

c) Delete(i): i là số, Phép Delete xoá ký tự tại vị trí i của xâu X.

Yêu cầu: Cho trước xâu Y, hãy tìm một số ít nhất các phép biến đổi trên để biến xâu X thành xâu Y.

Input: file văn bản STR.INP

Dòng 1: Chứa xâu X (độ dài ≤ 100)

Dòng 2: Chứa xâu Y (độ dài ≤ 100)

Output: file văn bản STR.OUT ghi các phép biến đổi cần thực hiện và xâu X tại mỗi phép biến đổi.

STR.INP	STR.OUT
PBBCEFATZQABCDABEFA	7 PBBCEFATZ -> Delete(9) -> PBBCEFAT PBBCEFAT -> Delete(8) -> PBBCEFA PBBCEFA -> Insert(4, B) -> PBBCBEFA PBBCBEFA -> Insert(4, A) -> PBBCABEFA PBBCABEFA -> Insert(4, D) -> PBBCDABEFA PBBCDABEFA -> Replace(2, A) -> PABCDABEFA PABCDABEFA -> Replace(1, Q) -> QABCDABEFA

Cách giải:

Đối với xâu ký tự thì việc xoá, chèn sẽ làm cho các phần tử phía sau vị trí biến đổi bị đánh chỉ số lại, gây khó khăn cho việc quản lý vị trí. Để khắc phục điều này, ta sẽ tìm một thứ tự biến đổi thoả mãn: Phép biến đổi tại vị trí i bắt buộc phải thực hiện sau các phép biến đổi tại vị trí i + 1, i + 2, …

Ví dụ: X = 'ABCD';

Insert(0, E) sau đó Delete(4) cho ra X = 'EABD'. Cách này không tuân thủ nguyên tắc

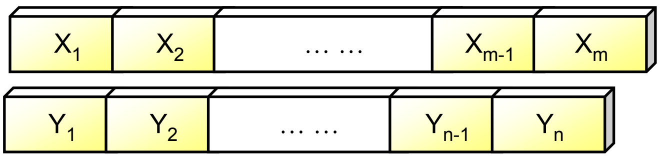
Delete(3) sau đó Insert(0, E) cho ra X = 'EABD'. Cách này tuân thủ nguyên tắc đề ra.

Nói tóm lại ta sẽ tìm một dãy biến đổi có vị trí thực hiện giảm dần.

3.3.1. Công thức truy hồi

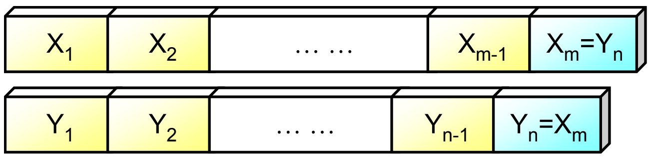
Giả sử m là độ dài xâu X và n là độ dài xâu Y. Gọi F[i, j] là số phép biến đổi tối thiểu để biến xâu gồm i ký tự đầu của xâu X: $X_1X_2 \dots X_i$ thành xâu gồm j ký tự đầu của xâu Y: $Y_1Y_2\dots Y_j$.

Quan sát hai dãy X và Y



Ta nhận thấy:

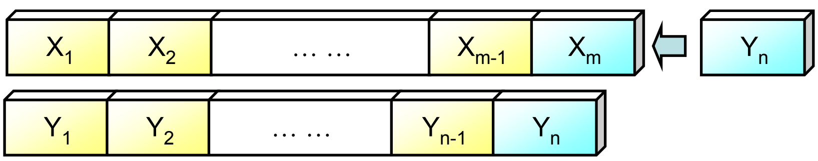
Nếu $X_m = Y_n$ thì ta chỉ cần biến đoạn $X_1X_2\dots X_{m-1}$ thành $Y_1Y_2\dots Y_{n-1}$



Tức là trong trường hợp này: **F[m, n] = F[m - 1, n - 1]**

Nếu $X_m \neq Y_n$ thì tại vị trí X_m ta có thể sử dụng một trong 3 phép biến đổi:

a) Hoặc chèn vào sau vị trí m của X, một ký tự đúng bằng Y_n:



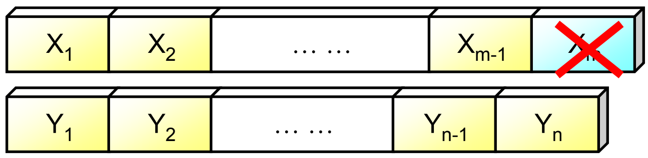
Thì khi đó F[m, n] sẽ bằng 1 phép chèn vừa rồi cộng với số phép biến đổi biến dãy X1…Xm thành dãy Y1…Yn-1: **F[m, n] = 1 + F[m, n - 1]**

b) Hoặc thay vị trí m của X bằng một ký tự đúng bằng Y_n:



Thì khi đó F[m, n] sẽ bằng 1 phép thay vừa rồi cộng với số phép biến đổi biến dãy $X_1…X_{m-1}$ thành dãy $Y_1…Y_{n-1}$: F[m, n] = 1 + F[m-1, n - 1]

c) Hoặc xoá vị trí thứ m của X:



Thì khi đó F[m, n] sẽ bằng 1 phép xoá vừa rồi cộng với số phép biến đổi biến dãy $X_1…X_{m-1}$ thành dãy $Y_1…Y_n$: F[m, n] = 1 + F[m-1, n]

Vì F[m, n] phải là nhỏ nhất có thể, nên trong trường hợp $X_m \neq Y_n$ thì

F[m, n] = min(F[m, n - 1], F[m - 1, n - 1], F[m - 1, n]) + 1.

Ta xây dựng xong công thức truy hồi.

3.3.2. Cơ sở quy hoạch động

F[0, j] là số phép biến đổi biến xâu rỗng thành xâu gồm j ký tự đầu của F. Nó cần tối thiểu j phép chèn: F[0, j] = j

F[i, 0] là số phép biến đổi biến xâu gồm i ký tự đầu của S thành xâu rỗng, nó cần tối thiểu i phép xoá: F[i, 0] = i

Vậy đầu tiên bảng phương án F (cỡ[0..m, 0..n]) được khởi tạo hàng 0 và cột 0 là cơ sở quy hoạch động. Từ đó dùng công thức truy hồi tính ra tất cả các phần tử bảng B.

Sau khi tính xong thì F[m, n] cho ta biết số phép biến đổi tối thiểu.

Truy vết:

Nếu $X_m = Y_n$ thì chỉ việc xét tiếp F[m - 1, n - 1].

Nếu không, xét 3 trường hợp:

Nếu F[m, n] = F[m, n - 1] + 1 thì phép biến đổi đầu tiên được sử dụng là: Insert(m, Y_n)

Nếu F[m, n] = F[m - 1, n - 1] + 1 thì phép biến đổi đầu tiên được sử dụng là: Replace(m, Y_n)

Nếu F[m, n] = F[m - 1, n] + 1 thì phép biến đổi đầu tiên được sử dụng là: Delete(m)

Đưa về bài toán với m, n nhỏ hơn truy vết tiếp cho tới khi về F[0, 0]

Ví dụ: X =' ABCD'; Y = 'EABD' bảng phương án là:

F	0	1	2	3	4
0	0	1	2	3	4
1	1	1	1	2	3
2	2	2	2	1	2
3	3	3	3	2	2
4	4	4	4	3	2

Hình 50: Truy vết

Lưu ý: khi truy vết, để tránh truy nhập ra ngoài bảng, nên tạo viền cho bảng.

P_3_03_4.PAS * Biến đổi xâu

```
program StrOpt;
const
  InputFile  = 'STR.INP';
  OutputFile = 'STR.OUT';
  max = 100;
var
  X, Y: String[2 * max];
  F: array[-1..max, -1..max] of Integer;
  m, n: Integer;

procedure Enter;
var
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, X); ReadLn(fi, Y);
  Close(fi);
  m := Length(X); n := Length(Y);
end;

function Min3(x, y, z: Integer): Integer; {Cho giá trị nhỏ nhất trong 3 giá trị x, y, z}
var
  t: Integer;
begin
  if x < y then t := x else t := y;
  if z < t then t := z;
  Min3 := t;
end;

procedure Optimize;
var
  i, j: Integer;
begin
  {Khởi tạo viền cho bảng phương án}
  for i := 0 to m do F[i, -1] := max + 1;
  for j := 0 to n do F[-1, j] := max + 1;
  {Lưu cơ sở quy hoạch động}
  for j := 0 to n do F[0, j] := j;
  for i := 1 to m do F[i, 0] := i;
  {Dùng công thức truy hồi tính toàn bảng phương án}
  for i := 1 to m do
    for j := 1 to n do
      if X[i] = Y[j] then F[i, j] := F[i - 1, j - 1]
      else F[i, j] := Min3(F[i, j - 1], F[i - 1, j - 1], F[i - 1, j]) + 1;
end;
```

```
procedure Trace; {Truy vết}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, F[m, n]); {F[m, n] chính là số ít nhất các phép biến đổi cần thực hiện}
  while (m <> 0) or (n <> 0) do {Vòng lặp kết thúc khi m = n = 0}
    if X[m] = Y[n] then {Hai ký tự cuối của 2 xâu giống nhau}
      begin
        Dec(m); Dec(n); {Chỉ việc truy chéo lên trên bảng phương án}
      end
    else {Tại đây cần một phép biến đổi}
      begin
        Write(fo, X, ' -> '); {In ra xâu X trước khi biến đổi}
        if F[m, n] = F[m, n - 1] + 1 then {Nếu đây là phép chèn}
          begin
            Write(fo, 'Insert(', m, ', ', Y[n], ')');
            Insert(Y[n], X, m + 1);
            Dec(n); {Truy sang phải}
          end
        else
          if F[m, n] = F[m - 1, n - 1] + 1 then {Nếu đây là phép thay}
            begin
              Write(fo, 'Replace(', m, ', ', Y[n], ')');
              X[m] := Y[n];
              Dec(m); Dec(n); {Truy chéo lên trên}
            end
          else {Nếu đây là phép xoá}
            begin
              Write(fo, 'Delete(', m, ')');
              Delete(X, m, 1);
              Dec(m); {Truy lên trên}
            end;
        WriteLn(fo, ' -> ', X); {In ra xâu X sau phép biến đổi}
      end;
  Close(fo);
end;

begin
  Enter;
  Optimize;
  Trace;
end.
```

Bài này giải với các xâu ≤ 100 ký tự, nếu lưu bảng phương án dưới dạng mảng cấp phát động thì có thể làm với các xâu 255 ký tự. (Tốt hơn nên lưu mỗi dòng của bảng phương án là một mảng cấp phát động 1 chiều). Hãy tự giải thích tại sao khi giới hạn độ dài dữ liệu là 100, lại phải khai báo X và Y là String[200] chứ không phải là String[100] ?.

3.4. DÃY CON CÓ TỔNG CHIA HẾT CHO K

Cho một dãy gồm n ($1 \le n \le 1000$) số nguyên dương $A_1, A_2, \ldots, A_n$ và số nguyên dương k ($k \le 50$). Hãy tìm dãy con gồm nhiều phần tử nhất của dãy đã cho sao cho tổng các phần tử của dãy con này chia hết cho k.

Input: file văn bản SUBSEQ.INP

- Dòng 1: Chứa số n
- Dòng 2: Chứa n số $A_1, A_2, \ldots, A_n$ cách nhau ít nhất một dấu cách

Output: file văn bản SUBSEQ.OUT

- Dòng 1: Ghi độ dài dãy con tìm được
- Các dòng tiếp: Ghi các phần tử được chọn vào dãy con
- Dòng cuối: Ghi tổng các phần tử của dãy con đó.

SUBSEQ.INP	SUBSEQ.OUT
10 5 1 6 11 5 10 15 20 2 4 9	8 a[10] = 9 a[9] = 4 a[7] = 20 a[6] = 15 a[5] = 10 a[4] = 5 a[3] = 11 a[2] = 6 Sum = 80

3.4.1. Cách giải 1

Đề bài yêu cầu chọn ra một số tối đa các phần tử trong dãy A để được một dãy có tổng chia hết cho k, ta có thể giải bài toán bằng phương pháp duyệt tổ hợp bằng quay lui có đánh giá nhánh cận nhằm giảm bớt chi phí trong kỹ thuật vét cạn. Dưới đây ta trình bày phương pháp quy hoạch động:

Nhận xét 1: Không ảnh hưởng đến kết quả cuối cùng, ta có thể đặt:

$$A_i := A_i \text{ mod } k \text{ với } \forall i: 1 \le i \le n$$

Nhận xét 2: Gọi S là tổng các phần tử trong mảng A, ta có thể thay đổi cách tiếp cận bài toán: thay vì tìm xem phải chọn ra một số tối đa những phần tử để có tổng chia hết cho k, ta sẽ chọn ra một số tối thiểu các phần tử có tổng đồng dư với S theo modul k. Khi đó chỉ cần loại bỏ những phần tử này thì những phần tử còn lại sẽ là kết quả.

Nhận xét 3: Số phần tử tối thiểu cần loại bỏ bao giờ cũng nhỏ hơn k

Thật vậy, giả sử số phần tử ít nhất cần loại bỏ là m và các phần tử cần loại bỏ là A_{i_1}, A_{i_2}, …, A_{i_m}. Các phần tử này có tổng đồng dư với S theo mô-đun k. Xét các dãy sau

Dãy 0 := () = Dãy rỗng (Tổng ≡ 0 (mod k))

Dãy 1 := (A_{i_1})

Dãy 2 := (A_{i_1}, A_{i_2})

Dãy 3 := (A_{i_1}, A_{i_2}, A_{i_3})

… …

Dãy m := (A_{i_1}, A_{i_2}, …, A_{i_m})

Như vậy có m + 1 dãy, nếu m ≥ k thì theo nguyên lý Dirichlet sẽ tồn tại hai dãy có tổng đồng dư theo mô-đun k. Giả sử đó là hai dãy:

$$A_{i_1} + A_{i_2} + \ldots + A_{i_p} \equiv A_{i_1} + A_{i_2} + \ldots + A_{i_p} + \mathbf{A_{i_{p+1}} + \ldots + A_{i_q}} \pmod k$$

Suy ra $\mathbf{A}_{i_{p+1}} + \ldots + \mathbf{A}_{i_q}$ chia hết cho k. Vậy ta có thể xoá hết các phần tử này trong dãy đã chọn mà vẫn được một dãy có tổng đồng dư với S theo modul k, mâu thuẫn với giả thiết là dãy đã chọn có số phần tử tối thiểu.

Công thức truy hồi:

Nếu ta gọi F[i, t] là số phần tử tối thiểu phải chọn trong dãy $A_1, A_2, \ldots, A_i$ để có tổng chia k dư t. Nếu không có phương án chọn ta coi F[i, t] = +∞ . Khi đó F[i, t] được tính qua công thức truy hồi sau:

Nếu trong dãy trên không phải chọn A_i thì F[i, t] = F[i - 1, t];

Nếu trong dãy trên phải chọn A_i thì F[i, t] = 1 + F[i - 1, $\overline{t - A_i}$] ($\overline{t - A_i}$ ở đây hiểu là phép trừ trên các lớp đồng dư mod k. Ví dụ khi k = 7 thì $\overline{1 - 3} = 5$)

Từ trên suy ra F[i, t] = min (F[i - 1, t], 1 + F[i - 1, t - A_i]).

Còn tất nhiên, cơ sở quy hoạch động: F(0, 0) = 0; F(0, i) = + ∞ (với ∀i: 1 ≤ i < k).

Bảng phương án F có kích thước [0..n, 0.. k - 1] tối đa là 1001x50 phần tử kiểu Byte.

P_3_03_5.PAS * Dãy con có tổng chia hết cho k

```
program SubSequence;
const
  InputFile  = 'SUBSEQ.INP';
  OutputFile = 'SUBSEQ.OUT';
  maxN = 1000;
  maxK = 50;
var
  a: array[1..maxN] of Integer;
  f: array[0..maxN, 0..maxK - 1] of Byte;
  n, k: Integer;

procedure Enter;
var
  fi: Text;
  i: Integer;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, k);
  for i := 1 to n do Read(fi, a[i]);
  Close(fi);
end;

function Sub(x, y: Integer): Integer; {Tính x - y (theo mod k)}
var
  tmp: Integer;
begin
  tmp := (x - y) mod k;
  if tmp >= 0 then Sub := tmp
  else Sub := tmp + k;
end;

procedure Optimize;
var
  i, t: Integer;
begin
  FillChar(f, SizeOf(f), $FF); {Khởi tạo các phần tử f[0, .] đều bằng 255 (+∞)}
  f[0, 0] := 0; {Ngoại trừ f[0, 0] := 0}
```

```
  for i := 1 to n do
    for t := 0 to k - 1 do {Tính f[i, t] := min (f[i - 1, t], f[i - 1, Sub(t, aᵢ)] + 1}
      if f[i - 1, t] < f[i - 1, Sub(t, a[i])] + 1 then
        f[i, t] := f[i - 1, t]
      else
        f[i, t] := f[i - 1, Sub(t, a[i])] + 1;
end;

procedure Result;
var
  fo: Text;
  i, t: Integer;
  SumAll, Sum: LongInt;
begin
  SumAll := 0;
  for i := 1 to n do SumAll := SumAll + a[i];
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, n - f[n, SumAll mod k]); {n - số phần tử bỏ đi = số phần tử giữ lại}
  i := n; t := SumAll mod k;
  Sum := 0;
  for i := n downto 1 do
    if f[i, t] = f[i - 1, t] then {Nếu phương án tối ưu không bỏ aᵢ, tức là có chọn aᵢ}
      begin
        WriteLn(fo, 'a[', i, '] = ', a[i]);
        Sum := Sum + a[i];
      end
    else
      t := Sub(t, a[i]);
  WriteLn(fo, 'Sum = ', Sum);
  Close(fo);
end;

begin
  Enter;
  Optimize;
  Result;
end.
```

3.4.2. Cách giải 2

Phân các phần tử trong dãy a theo các lớp đồng dư modul k. Lớp i gồm các phần tử chia k dư i. Gọi Count[i] là số lượng các phần tử thuộc lớp i.

Với $0 \le i, t < k$; Gọi f[i, t] là số phần tử nhiều nhất có thể chọn được trong các lớp 0, 1, 2, …, i để được tổng chia k dư t. Trong trường hợp có cách chọn, gọi Trace[i, t] là số phần tử được chọn trong lớp i theo phương án này, trong trường hợp không có cách chọn, Trace[i, t] được coi là -1.

Ta dễ thấy rằng f[0, 0] = Count[0], Trace[0, 0] = Count[0], còn Trace[0, i] với $i \neq 0$ bằng -1.

Với $i \ge 1$; $0 \le t < k$, nếu có phương án chọn ra nhiều phần tử nhất trong các lớp từ 0 tới i để được tổng chia k dư t thì phương án này có thể chọn j phần tử của lớp i ($0 \le j \le$ Count[i]), nếu bỏ j phần tử này đi, sẽ phải thu được phương án chọn ra nhiều phần tử nhất trong các lớp từ 0 tới i - 1 để được tổng chia k dư $\overline{t - i * j}$. Từ đó suy ra công thức truy hồi:

$$f[i,t] = \max_{\substack{0 \le j \le Count[i] \\ Trace[i-1, \overline{t-j*i}) \ne -1}} (f[i-1, \overline{t-j*i}] + j)$$

$$Trace[i,t] = \arg\max_{\substack{0 \le j \le Count[i] \\ Trace[i-1, \overline{t-j*i}) \ne -1}} (f[i-1, \overline{t-j*i}] + j)$$

P_3_03_6.PAS * Dãy con có tổng chia hết cho k

```
program SubSequence;
const
  InputFile  = 'SUBSEQ.INP';
  OutputFile = 'SUBSEQ.OUT';
  maxN = 1000;
  maxK = 50;
var
  a: array[1..maxN] of Integer;
  Count: array[0..maxK - 1] of Integer;
  f, Trace: array[0..maxK - 1, 0..maxK - 1] of Integer;
  n, k: Integer;

procedure Enter;
var
  fi: Text;
  i: Integer;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, k);
  FillChar(Count, SizeOf(Count), 0);
  for i := 1 to n do
    begin
      Read(fi, a[i]);
      Inc(Count[a[i] mod k]); {Nhập dữ liệu đồng thời với việc tính các Count[.]}
    end;
  Close(fi);
end;

function Sub(x, y: Integer): Integer;
var
  tmp: Integer;
begin
  tmp := (x - y) mod k;
  if tmp >= 0 then Sub := tmp
  else Sub := tmp + k;
end;

procedure Optimize;
var
  i, j, t: Integer;
begin
  FillChar(f, SizeOf(f), 0);
  f[0, 0] := Count[0];
  FillChar(Trace, SizeOf(Trace), $FF); {Khởi tạo các mảng Trace=-1}
  Trace[0, 0] := Count[0]; {Ngoại trừ Trace[0, 0] = Count[0]}
  for i := 1 to k - 1 do
    for t := 0 to k - 1 do
      for j := 0 to Count[i] do
        if (Trace[i - 1, Sub(t, j * i)] <> -1) and
           (f[i, t] < f[i - 1, Sub(t, j * i)] + j) then
          begin
            f[i, t] := f[i - 1, Sub(t, j * i)] + j;
            Trace[i, t] := j;
          end;
end;
```

```
procedure Result;
var
  fo: Text;
  i, t, j: Integer;
  Sum: LongInt;
begin
  t := 0;
  {Tính lại các Count[i] := Số phần tử phương án tối ưu sẽ chọn trong lớp i}
  for i := k - 1 downto 0 do
    begin
      j := Trace[i, t];
      t := Sub(t, j * i);
      Count[i] := j;
    end;
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, f[k - 1, 0]);
  Sum := 0;
  for i := 1 to n do
    begin
      t := a[i] mod k;
      if Count[t] > 0 then
        begin
          WriteLn(fo, 'a[', i, '] = ', a[i]);
          Dec(Count[t]);
          Sum := Sum + a[i];
        end;
    end;
  WriteLn(fo, 'Sum = ', Sum);
  Close(fo);
end;

begin
  Enter;
  Optimize;
  Result;
end.
```

Cách giải thứ hai tốt hơn cách giải thứ nhất vì nó có thể thực hiện với n lớn. Ví dụ này cho thấy một bài toán quy hoạch động có thể có nhiều cách đặt công thức truy hồi để giải.

3.5. PHÉP NHÂN TỔ HỢP DÃY MA TRẬN

Với ma trận A kích thước pxq và ma trận B kích thước qxr. Người ta có phép nhân hai ma trận đó để được ma trận C kích thước pxr. Mỗi phần tử của ma trận C được tính theo công thức:

$$C_{ij} = \sum_{k=1}^{q} A_{ik}.B_{kj}; \quad 1 \le i \le p, 1 \le j \le r$$

Ví dụ:

A là ma trận kích thước 3x4, B là ma trận kích thước 4x5 thì C sẽ là ma trận kích thước 3x5

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} x \begin{bmatrix} 1 & 0 & 2 & 4 & 0 \\ 0 & 1 & 0 & 5 & 1 \\ 3 & 0 & 1 & 6 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 14 & 6 & 9 & 36 & 9 \\ 34 & 14 & 25 & 100 & 21 \\ 54 & 22 & 41 & 164 & 33 \end{bmatrix}$$

Để thực hiện phép nhân hai ma trận A(mxn) và B(nxp) ta có thể làm như đoạn chương trình sau:

```
for i := 1 to p do
  for j := 1 to r do
    begin
      cij := 0;
      for k := 1 to q do cij := cij + aik * bkj;
    end;
```

Phí tổn để thực hiện phép nhân này có thể đánh giá qua số phép nhân, để nhân hai ma trận A(pxq) và B(qxr) ta cần thực hiện p.q.r phép nhân số học.

Phép nhân ma trận không có tính chất giao hoán nhưng có tính chất kết hợp

$$(A * B) * C = A * (B * C)$$

Vậy nếu A là ma trận cấp 3x4, B là ma trận cấp 4x10 và C là ma trận cấp 10x15 thì:

Để tính (A * B) * C, ta thực hiện (A * B) trước, được ma trận X kích thước 3x10 sau 3.4.10 = 120 phép nhân số. Sau đó ta thực hiện X * C được ma trận kết quả kích thước 3x15 sau 3.10.15 = 450 phép nhân số. Vậy tổng số phép nhân số học phải thực hiện sẽ là 570.

Để tính A * (B * C), ta thực hiện (B * C) trước, được ma trận Y kích thước 4x15 sau 4.10.15 = 600 phép nhân số. Sau đó ta thực hiện A * Y được ma trận kết quả kích thước 3x15 sau 3.4.15 = 180 phép nhân số. Vậy tổng số phép nhân số học phải thực hiện sẽ là 780.

Vậy thì trình tự thực hiện có ảnh hưởng lớn tới chi phí. Vấn đề đặt ra là tính số phí tổn ít nhất khi thực hiện phép nhân một dãy các ma trận:

$$M_1 * M_2 * \ldots * M_n$$

Với :

M_1 là ma trận kích thước a_1 x a_2

M_2 là ma trận kích thước a_2 x a_3

…

M_n là ma trận kích thước a_n x a_{n+1}

Input: file văn bản MULTMAT.INP

- Dòng 1: Chứa số nguyên dương $n \le 100$
- Dòng 2: Chứa n + 1 số nguyên dương $a_1, a_2, \ldots, a_{n+1}$ ($\forall i$: $1 \le a_i \le 100$) cách nhau ít nhất một dấu cách

Output: file văn bản MULTMAT.OUT

- Dòng 1: Ghi số phép nhân số học tối thiểu cần thực hiện
- Dòng 2: Ghi biểu thức kết hợp tối ưu của phép nhân dãy ma trận

MULTMAT.INP	MULTMAT.OUT
6 3 2 3 1 2 2 3	31 ((M[1] * (M[2] * M[3])) * ((M[4] * M[5]) * M[6]))

Trước hết, nếu dãy chỉ có một ma trận thì chi phí bằng 0, tiếp theo ta nhận thấy để nhân một cặp ma trận thì không có chuyện kết hợp gì ở đây cả, chi phí cho phép nhân đó là tính được ngay. Vậy thì phí tổn cho phép nhân hai ma trận liên tiếp trong dãy là hoàn toàn có thể ghi nhận lại được. Sử dụng những thông tin đã ghi nhận để tối ưu hoá phí tổn nhân những bộ ba ma trận liên tiếp … Cứ tiếp tục như vậy cho tới khi ta tính được phí tổn nhân n ma trận liên tiếp.

3.5.1. Công thức truy hồi:

Gọi F[i, j] là số phép nhân tối thiểu cần thực hiện để nhân đoạn ma trận liên tiếp: $M_i*M_{i+1}*…*M_j$. Thì khi đó F[i, i] = 0 với $\forall$i.

Để tính $M_i * M_{i+1} * … * M_j$, ta có thể có nhiều cách kết hợp:

$M_i * M_{i+1} * … * M_j = (M_i * M_{i+1} * … * M_k) * (M_{k+1} * M_{k+2} * … * M_j)$ (Với $i \le k < j$)

Với một cách kết hợp (phụ thuộc vào cách chọn vị trí k), chi phí tối thiểu phải thực hiện bằng:

Chi phí thực hiện phép nhân $M_i * M_{i+1} * … * M_k$ = F[i, k]

Cộng với chi phí thực hiện phép nhân $M_{k+1} * M_{k+2} * … * M_j$ = F[k + 1, j]

Cộng với chi phí thực hiện phép nhân hai ma trận cuối cùng: ma trận tạo thành từ phép nhân $(M_i * M_{i+1} * … * M_k)$ có kích thước a_i x a_{k+1} và ma trận tạo thành từ phép nhân $(M_{k+1} * M_{k+2} * … * M_j)$ có kích thước a_{k+1} x a_{j+1}, vậy chi phí này là $a_i * a_{k+1} * a_{j+1}$.

Từ đó suy ra: do có nhiều cách kết hợp, mà ta cần chọn cách kết hợp để có chi phí ít nhất nên ta sẽ cực tiểu hoá F[i, j] theo công thức:

$$F[i, j] = \min_{i \le k < j}(F[i, k] + F[k+1, j] + a_i * a_{k+1} * a_{j+1})$$

3.5.2. Tính bảng phương án

Bảng phương án F là bảng hai chiều, nhìn vào công thức truy hồi, ta thấy F[i, j] chỉ được tính khi mà F[i, k] cũng như F[k + 1, j] đều đã biết. Tức là ban đầu ta điền cơ sở quy hoạch động vào đường chéo chính của bảng(F[i, i] = 0), từ đó tính các giá trị thuộc đường chéo nằm phía trên (Tính các F[i, i + 1]), rồi lại tính các giá trị thuộc đường chéo nằm phía trên nữa (F[i, i + 2]) … Đến khi tính được F[1, n] thì dừng lại

3.5.3. Tìm cách kết hợp tối ưu

Tại mỗi bước tính F[i, j], ta ghi nhận lại điểm k mà cách tính $(M_i * M_{i+1} * … * M_k) * (M_{k+1} * M_{k+2} * … * M_j)$ cho số phép nhân số học nhỏ nhất, chẳng hạn ta đặt T[i, j] = k.

Khi đó, muốn in ra phép kết hợp tối ưu để nhân đoạn $M_i * M_{i+1} * … * M_k * M_{k+1} * M_{k+2} * … * M_j$, ta sẽ in ra cách kết hợp tối ưu để nhân đoạn $M_i * M_{i+1} * … * M_k$ và cách kết hợp tối ưu

để nhân đoạn $M_{k+1} * M_{k+2} * \ldots * M_j$ (có kèm theo dấu đóng mở ngoặc) đồng thời viết thêm dấu "*" vào giữa hai biểu thức đó.

P_3_03_7.PAS * Nhân tối ưu dãy ma trận

```
program MatrixesMultiplier;
const
  InputFile  = 'MULTMAT.INP';
  OutputFile = 'MULTMAT.OUT';
  max = 100;
  MaxLong = 1000000000;
var
  a: array[1..max + 1] of Integer;
  F: array[1..max, 1..max] of LongInt;
  T: array[1..max, 1..max] of Byte;
  n: Integer;
  fo: Text;

procedure Enter; {Nhập dữ liệu từ thiết bị nhập chuẩn}
var
  i: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n);
  for i := 1 to n + 1 do Read(fi, a[i]);
  Close(fi);
end;

procedure Optimize;
var
  i, j, k, len: Integer;
  x, p, q, r: LongInt;
begin
  for i := 1 to n do
    for j := i to n do
      if i = j then F[i, j] := 0
      else F[i, j] := MaxLong; {Khởi tạo bảng phương án: đường chéo chính = 0, các ô khác = +∞}
  for len := 2 to n do {Tìm cách kết hợp tối ưu để nhân đoạn gồm len ma trận liên tiếp}
    for i := 1 to n - len + 1 do
      begin
        j := i + len - 1; {Tính F[i, j]}
        for k := i to j - 1 do {Xét mọi vị trí phân hoạch k}
          begin {Giả sử ta tính Mi * … * Mj = (Mi * … * Mk) * (Mk+1 * … * Mj)}
            p := a[i]; q := a[k + 1]; r := a[j + 1]; {Kích thước 2 ma trận sẽ nhân cuối cùng}
            x := F[i, k] + F[k + 1, j] + p * q * r; {Chi phí nếu phân hoạch theo k}
            if x < F[i, j] then {Nếu phép phân hoạch đó tốt hơn F[i, j] thì ghi nhận lại}
              begin
                F[i, j] := x;
                T[i, j] := k;
              end;
          end;
      end;
end;

procedure Trace(i, j: Integer); {In ra phép kết hợp để nhân đoạn Mi * Mi+1 * … * Mj}
var
 k: Integer;
begin
  if i = j then Write(fo, 'M[', i, ']') {Nếu đoạn chỉ gồm 1 ma trận thì in luôn}
  else {Nếu đoạn gồm từ 2 ma trận trở lên}
    begin
      Write(fo, '('); {Mở ngoặc}
```

```
      k := T[i, j]; {Lấy vị trí phân hoạch tối ưu đoạn Mi…Mj}
      Trace(i, k); {In ra phép kết hợp để nhân đoạn đầu}
      Write(fo, ' * '); {Dấu nhân}
      Trace(k + 1, j); {In ra phép kết hợp để nhân đoạn sau}
      Write(fo, ')'); {Đóng ngoặc}
    end;
end;

begin
  Enter;
  Optimize;
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, F[1, n]); {Số phép nhân cần thực hiện}
  Trace(1, n); {Truy vết bằng đệ quy}
  Close(fo);
end.
```

3.6. BÀI TẬP LUYỆN TẬP

3.6.1. Bài tập có gợi ý lời giải

Bài 1

Nhập vào hai số nguyên dương n và k (n, k ≤ 100). Hãy cho biết

a) Có bao nhiêu số nguyên dương có ≤ n chữ số mà tổng các chữ số đúng bằng k. Nếu có hơn 1 tỉ số thì chỉ cần thông báo có nhiều hơn 1 tỉ.

b) Nhập vào một số p ≤ 1 tỉ. Cho biết nếu đem các số tìm được xếp theo thứ tự tăng dần thì số thứ p là số nào ?

Gợi ý:

Câu a: Ta sẽ đếm số các số có đúng n chữ số mà tổng các chữ số (TCCS) bằng k, chỉ có điều các số của ta cho phép có thể bắt đầu bằng 0. Ví dụ: ta coi 0045 là số có 4 chữ số mà TCCS là 9. Gọi F[n, k] là số các số có n chữ số mà TCCS bằng k. Các số đó có dạng $\overline{x_1x_2...x_n}$; x1, x2, …xn ở đây là các chữ số 0…9 và x1 + x2 + … + xn = k. Nếu cố định x1 = t thì ta nhận thấy $\overline{x_2...x_n}$ lập thành một số có n - 1 chữ số mà TCCS bằng k - t. Suy ra do x1 có thể nhận các giá trị từ 0 tới 9 nên về mặt số lượng: F[n, k] = $\sum_{t=0}^{9} F[n-1, k-t]$. Đây là công thức truy hồi tính F[n, k], thực ra chỉ xét những giá trị t từ 0 tới 9 và t ≤ k mà thôi (để tránh trường hợp k - t <0). Chú ý rằng nếu tại một bước nào đó tính ra một phần tử của F > 10^9 thì ta đặt lại phần tử đó là 10^9 + 1 để tránh bị tràn số do cộng hai số quá lớn. Kết thúc quá trình tính toán, nếu F[n, k] = 10^9 + 1 thì ta chỉ cần thông báo chung chung là có > 1 tỉ số.

Cơ sở quy hoạch động thì có thể đặt là:

F[1, k] = số các số có 1 chữ số mà TCCS bằng k, như vậy nếu k ≥ 10 thì F[1, k] = 0 còn nếu 0 ≤ k ≤ 9 thì F[1, k] = 1.

Câu b: Dựa vào bảng phương án F[0..n, 0..k],

F[n - 1, k] = số các số có n - 1 CS mà TCCS bằng k = số các số có n CS, bắt đầu là 0, TCCS bằng k.

F[n - 1, k - 1] = số các số có n - 1 CS mà TCCS bằng k - 1 = số các số có n CS, bắt đầu là 1, TCCS bằng k.

F[n - 1, k - 2] = số các số có n - 1 CS mà TCCS bằng k - 2 = số các số có n CS, bắt đầu là 2, TCCS bằng k.

…

F[n - 1, k - 9] = số các số có n - 1 CS mà TCCS bằng k - 9 = số các số có n CS, bắt đầu là 9, TCCS bằng k.

Từ đó ta có thể biết được số thứ p (theo thứ tự tăng dần) cần tìm sẽ có chữ số đầu tiên là chữ số nào, tương tự ta sẽ tìm được chữ số thứ hai, thứ ba v.v… của số đó.

Bài 2

Cho n gói kẹo ($n \le 200$), mỗi gói chứa không quá 200 viên kẹo, và một số $M \le 40000$. Hãy chỉ ra một cách lấy ra một số các gói kẹo để được tổng số kẹo là M, hoặc thông báo rằng không thể thực hiện được việc đó.

Gợi ý: Giả sử số kẹo chứa trong gói thứ i là A_i

Gọi b[V] là số nguyên dương **bé nhất** thoả mãn: Có thể chọn trong số các gói kẹo từ gói 1 đến gói b[V] ra một số gói để được tổng số kẹo là V. Nếu không có phương án chọn, ta coi $b[V] = +\infty$. Trước tiên, khởi tạo b[0] = 0 và các $b[V] = +\infty$ với mọi $V > 0$. Ta sẽ xây dựng b[V] như sau:

Để tiện nói, ta đặt k = b[V]. Vì k là bé nhất có thể, nên nếu có cách chọn trong số các gói kẹo từ gói 1 đến gói k để được số kẹo V thì **chắc chắn phải chọn gói k**. Mà đã chọn gói k rồi thì **trong số các gói kẹo từ 1 đến k - 1**, phải chọn ra được một số gói **để được số kẹo là V - A_k**. Tức là $b[V - A_k] \le k - 1 < k$. Vậy thì b[V] sẽ được tính bằng cách:

Xét tất cả các gói kẹo k có $A_k \le V$ và thoả mãn $b[V - A_k] < k$, chọn ra chỉ số k bé nhất, sau đó gán b[V] := k. Đây chính là công thức truy hồi tính bảng phương án.

Sau khi đã tính b[1], b[2], …, b[M]. Nếu b[M] vẫn bằng $+\infty$ thì có nghĩa là không có phương án chọn. Nếu không thì sẽ chọn gói p_1 = b[M], tiếp theo sẽ chọn gói $p_2 = b[M - A_{p_1}]$, rồi lại chọn gói $p3 = b[M - Ap_1 - Ap_2]$… Đến khi truy vết về tới b[0] thì thôi.

Bài 3

Cho n gói kẹo ($n \le 200$), mỗi gói chứa không quá 200 viên kẹo, hãy chia các gói kẹo ra làm hai nhóm sao cho số kẹo giữa hai nhóm chênh lệch nhau ít nhất

Gợi ý:

Gọi S là tổng số kẹo và M là nửa tổng số kẹo, áp dụng cách giải như bài 2. Sau đó

Tìm số nguyên dương T thoả mãn:

- $T \le M$
- Tồn tại một cách chọn ra một số gói kẹo để được tổng số kẹo là T ($b[T] \ne +\infty$)
- T lớn nhất có thể

Sau đó chọn ra một số gói kẹo để được T viên kẹo, các gói kẹo đó được đưa vào một nhóm, số còn lại vào nhóm thứ hai.

Bài 4

Cho một bảng A kích thước m x n, trên đó ghi các số nguyên. Một người xuất phát tại ô nào đó của cột 1, cần sang cột n (tại ô nào cũng được). Quy tắc: Từ ô A[i, j] chỉ được quyền sang một trong 3 ô A[i, j + 1]; A[i - 1, j + 1]; A[i + 1, j + 1]. Hãy tìm vị trí ô xuất phát và hành trình đi từ cột 1 sang cột n sao cho tổng các số ghi trên đường đi là lớn nhất.

A =

1	2	6	7	9
7	6	5	6	7
1	2	3	4	2
4	7	8	7	6

Gợi ý:

Gọi B[i, j] là số điểm lớn nhất có thể có được khi tới ô A[i, j]. Rõ ràng đối với những ô ở cột 1 thì B[i, 1] = A[i, 1]:

A =

1	2	6	7	9
7	6	5	6	7
1	2	3	4	2
4	7	8	7	6

B =

1				
7				
1				
4				

Với những ô (i, j) ở các cột khác. Vì chỉ những ô (i, j - 1), (i - 1, j - 1), (i + 1, j - 1) là có thể sang được ô (i, j), và khi sang ô (i, j) thì số điểm được cộng thêm A[i, j] nữa. Chúng ta cần B[i, j] là số điểm lớn nhất có thể nên B[i, j] = max(B[i, j - 1], B[i - 1, j - 1], B[i + 1, j - 1]) + A[i, j]. Ta dùng công thức truy hồi này tính tất cả các B[i, j]. Cuối cùng chọn ra B[i, n] là phần tử lớn nhất trên cột n của bảng B và từ đó truy vết tìm ra đường đi nhiều điểm nhất.

3.6.2. Bài tập tự làm

Bài 1

Bài toán cái túi với kích thước như nêu trên là không thực tế, chẳng có siêu thị nào có ≤ 100 gói hàng cả. Hãy lập chương trình giải bài toán cái túi với n ≤ 10000; M ≤ 1000.

Bài 2

Xâu ký tự S gọi là xâu con của xâu ký tự T nếu có thể xoá bớt một số ký tự trong xâu T để được xâu S. Lập chương trình nhập vào hai xâu ký tự S_1, S_2. Tìm xâu S_3 có độ dài lớn nhất là xâu con của cả S_1 và S_2. Ví dụ: S_1 = 'abcdefghi123'; S_2 = 'abc1def2ghi3' thì S_3 là 'abcdefghi3'.

Bài 3

Một **xâu ký tự X gọi là chứa xâu ký tự Y** nếu như có thể xoá bớt một số ký tự trong xâu X để được xâu Y: Ví dụ: Xâu '1a2b3c45d' chứa xâu '12345'. Một xâu ký tự gọi là đối xứng nếu

nó không thay đổi khi ta viết các ký tự trong xâu theo thứ tự ngược lại: Ví dụ: 'abcABADABAcba', 'MADAM' là các xâu đối xứng.

Nhập một xâu ký tự S có độ dài không quá 128, hãy tìm xâu ký tự T thoả mãn cả 3 điều kiện:

1. Đối xứng
2. Chứa xâu S
3. Có ít ký tự nhất (có độ dài ngắn nhất)

Nếu có nhiều xâu T thoả mãn đồng thời 3 điều kiện trên thì chỉ cần cho biết một. Chẳng hạn với S = 'a_101_b' thì chọn T = 'ab_101_ba' hay T = 'ba_101_ab' đều đúng.

Ví dụ:

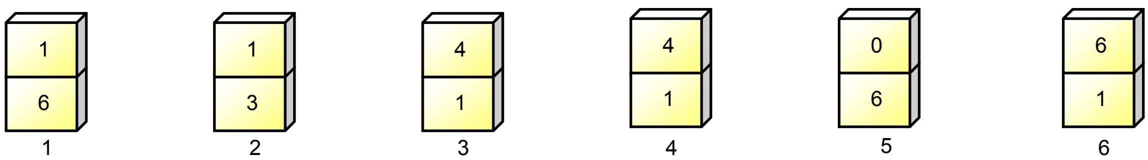
S	T
MADAM	MADAM
Edbabcd	**ed**c**babcd**e
00_11_22_33_222_1_000	000_11_222_33_222_11_000
abcdefg_hh_gfe_1_d_2_c_3_ba	ab_3_c_2_d_1_efg_hh_gfe_1_d_2_c_3_ba

Bài 4

Có n loại tiền giấy: Tờ giấy bạc loại i có mệnh giá là V[i] (n ≤ 20, 1 ≤ V[i] ≤ 10000). Hỏi muốn mua một món hàng giá là M thì có bao nhiêu cách trả số tiền đó bằng những loại giấy bạc đã cho (Trường hợp có > 1 tỉ cách thì chỉ cần thông báo có nhiều hơn 1 tỉ). Nếu tồn tại cách trả, cho biết cách trả phải dùng ít tờ tiền nhất.

Bài 5

Cho n quân đô-mi-nô xếp dựng đứng theo hàng ngang và được đánh số từ 1 đến n. Quân đô-mi-nô thứ i có số ghi ở ô trên là a[i] và số ghi ở ô dưới là b[i]. Xem hình vẽ:



Biết rằng $1 \le n \le 100$ và $0 \le a_i, b_i \le 6$ với $\forall i: 1 \le i \le n$. Cho phép lật ngược các quân đô-mi-nô. Khi một quân đô-mi-nô thứ i bị lật, nó sẽ có số ghi ở ô trên là b[i] và số ghi ở ô dưới là a[i].

Vấn đề đặt ra là hãy tìm cách lật các quân đô-mi-nô sao cho chênh lệch giữa tổng các số ghi ở hàng trên và tổng các số ghi ở hàng dướii là tối thiểu. Nếu có nhiều phương án lật tốt như nhau, thì chỉ ra phương án phải lật ít quân nhất.

Như ví dụ trên thì sẽ lật hai quân Đô-mi-nô thứ 5 và thứ 6. Khi đó:

Tổng các số ở hàng trên = 1 + 1 + 4 + 4 + 6 + 1 = 17

Tổng các số ở hàng dưới = 6 + 3 + 1 + 1 + 0 + 6 = 17

Bài 6

Xét bảng H kích thước 4x4, các hàng và các cột được đánh chỉ số A, B, C, D. Trên 16 ô của bảng, mỗi ô ghi 1 ký tự A hoặc B hoặc C hoặc D.

	A	B	C	D
A	A	A	B	B
B	C	D	A	B
C	B	C	B	A
D	B	D	D	D

Cho xâu S gồm n ký tự chỉ gồm các chữ A, B, C, D.

Xét phép co R(i): thay ký tự S_i và S_{i+1} bởi ký tự nằm trên hàng S_i, cột S_{i+1} của bảng H.

Ví dụ: S = ABCD; áp dụng liên tiếp 3 lần R(1) sẽ được

ABCD → ACD → BD → B.

Yêu cầu: Cho trước một ký tự X∈{A, B, C, D}, hãy chỉ ra thứ tự thực hiện n - 1 phép co để ký tự còn lại cuối cùng trong S là X.

Bài 7

Cho N số tự nhiên A_1, A_2, …, A_N. Biết rằng $1 \le N \le 200$ và $0 \le A_i \le 200$. Ban đầu các số được đặt liên tiếp theo đúng thứ tự cách nhau bởi dấu "?": A_1 ? A_2 ? … ? A_N. Yêu cầu: Cho trước số nguyên K, hãy tìm cách thay các dấu "?" bằng dấu cộng hay dấu trừ để được một biểu thức số học cho giá trị là K. Biết rằng $1 \le N \le 200$ và $0 \le A_i \le 100$.

Ví dụ: Ban đầu 1 ? 2 ? 3 ? 4 và K = 0 sẽ cho kết quả 1 - 2 - 3 + 4.

Bài 8

Dãy Catalan là một dãy số tự nhiên bắt đầu là 0, kết thúc là 0, hai phần tử liên tiếp hơn kém nhau 1 đơn vị. Hãy lập chương trình nhập vào số nguyên dương n lẻ và một số nguyên dương p. Cho biết rằng nếu như ta đem tất cả các dãy Catalan độ dài n xếp theo thứ tự từ điển thì dãy thứ p là dãy nào.

Một bài toán quy hoạch động có thể có nhiều cách tiếp cận khác nhau, chọn cách nào là tuỳ theo yêu cầu bài toán sao cho dễ dàng cài đặt nhất. Phương pháp này thường không khó khăn trong việc tính bảng phương án, không khó khăn trong việc tìm cơ sở quy hoạch động, mà khó khăn chính là **nhìn nhận ra bài toán quy hoạch động** và **tìm ra công thức truy hồi** giải nó, công việc này đòi hỏi sự nhanh nhạy, khôn khéo, mà chỉ từ sự rèn luyện mới có thể có được. Hãy đọc lại §1 để tìm hiểu kỹ các phương pháp thông dụng khi cài đặt một chương trình giải công thức truy hồi.

PHẦN 4. CÁC THUẬT TOÁN TRÊN ĐỒ THỊ



Leonhard Euler
(1707-1783)

Trên thực tế có nhiều bài toán liên quan tới một tập các đối tượng và những mối liên hệ giữa chúng, đòi hỏi toán học phải đặt ra một mô hình biểu diễn một cách chặt chẽ và tổng quát bằng ngôn ngữ ký hiệu, đó là đồ thị. Những ý tưởng cơ bản của nó được đưa ra từ thế kỷ thứ XVIII bởi nhà toán học Thuỵ Sĩ Leonhard Euler, ông đã dùng mô hình đồ thị để giải bài toán về những cây cầu Konigsberg nổi tiếng.

Mặc dù Lý thuyết đồ thị đã được khoa học phát triển từ rất lâu nhưng lại có nhiều ứng dụng hiện đại. Đặc biệt trong khoảng vài mươi năm trở lại đây, cùng với sự ra đời của máy tính điện tử và sự phát triển nhanh chóng của Tin học, Lý thuyết đồ thị càng được quan tâm đến nhiều hơn. Đặc biệt là các thuật toán trên đồ thị đã có nhiều ứng dụng trong nhiều lĩnh vực khác nhau như: Mạng máy tính, Lý thuyết mã, Tối ưu hoá, Kinh tế học v.v… Hiện nay, môn học này là một trong những kiến thức cơ sở của bộ môn khoa học máy tính.

Trong phạm vi một chuyên đề, không thể nói kỹ và nói hết những vấn đề của lý thuyết đồ thị. Tập bài giảng này sẽ xem xét lý thuyết đồ thị dưới góc độ người lập trình, tức là khảo sát những **thuật toán cơ bản nhất** có thể **dễ dàng cài đặt trên máy tính** một số ứng dụng của nó. . Công việc của người lập trình là đọc hiểu được ý tưởng cơ bản của thuật toán và cài đặt được chương trình trong bài toán tổng quát cũng như trong trường hợp cụ thể.

§1. CÁC KHÁI NIỆM CƠ BẢN

1.1. ĐỊNH NGHĨA ĐỒ THỊ (GRAPH)

Là một cấu trúc rời rạc gồm các đỉnh và các cạnh nối các đỉnh đó. Được mô tả hình thức:

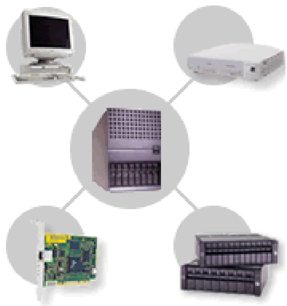
$$G = (V, E)$$

V gọi là tập các **đỉnh** (Vertices) và E gọi là tập các **cạnh** (Edges). Có thể coi E là tập các cặp (u, v) với u và v là hai đỉnh của V.

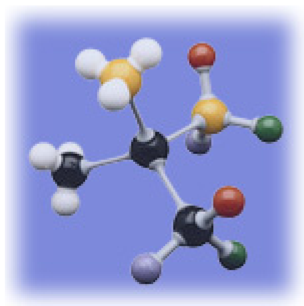
Một số hình ảnh của đồ thị:



Sơ đồ giao thông



Mạng máy tính



Cấu trúc phân tử

Hình 51: Ví dụ về mô hình đồ thị

Có thể phân loại đồ thị theo đặc tính và số lượng của tập các cạnh E:

Cho đồ thị G = (V, E). Định nghĩa một cách hình thức

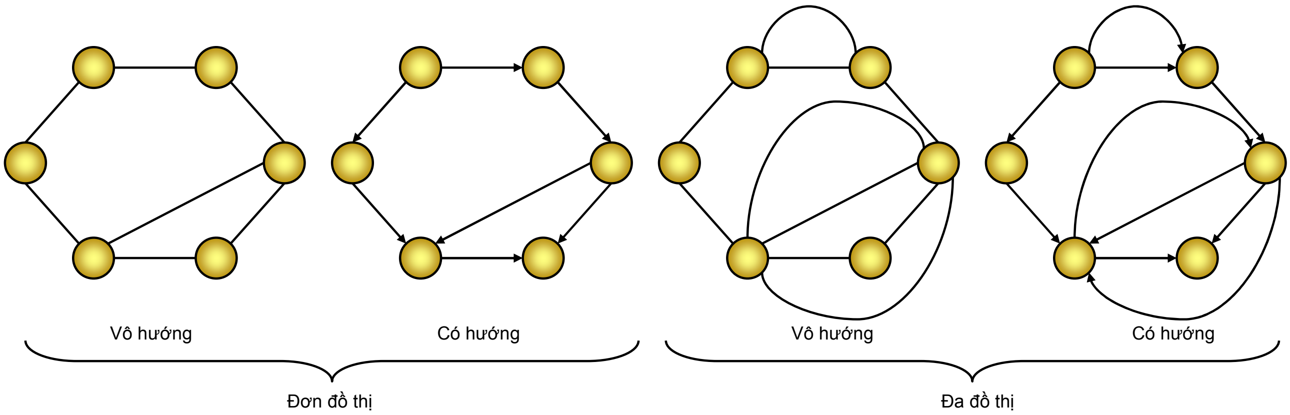
G được gọi là **đơn đồ thị** nếu giữa hai đỉnh u, v của V có nhiều nhất là 1 cạnh trong E nối từ u tới v.

G được gọi là **đa đồ thị** nếu giữa hai đỉnh u, v của V có thể có nhiều hơn 1 cạnh trong E nối từ u tới v (Hiển nhiên đơn đồ thị cũng là đa đồ thị).

G được gọi là đồ thị **vô hướng** (undirected graph) nếu các cạnh trong E là không định hướng, tức là cạnh nối hai đỉnh u, v bất kỳ cũng là cạnh nối hai đỉnh v, u. Hay nói cách khác, tập E gồm các cặp (u, v) không tính thứ tự. $(u, v) \equiv (v, u)$

G được gọi là đồ thị **có hướng** (directed graph) nếu các cạnh trong E là có định hướng, có thể có cạnh nối từ đỉnh u tới đỉnh v nhưng chưa chắc đã có cạnh nối từ đỉnh v tới đỉnh u. Hay nói cách khác, tập E gồm các cặp (u, v) có tính thứ tự: $(u, v) \neq (v, u)$. Trong đồ thị có hướng, các cạnh được gọi là các **cung**. Đồ thị vô hướng cũng có thể coi là đồ thị có hướng nếu như ta coi cạnh nối hai đỉnh u, v bất kỳ tương đương với hai cung (u, v) và (v, u).

Ví dụ:



Hình 52: Phân loại đồ thị

1.2. CÁC KHÁI NIỆM

Như trên định nghĩa **đồ thị G = (V, E) là một cấu trúc rời rạc**, tức là các tập V và E hoặc là tập hữu hạn, hoặc là tập đếm được, có nghĩa là ta có thể đánh số thứ tự 1, 2, 3… cho các phần tử của tập V và E. Hơn nữa, đứng trên phương diện người lập trình cho máy tính thì ta chỉ quan tâm đến các đồ thị hữu hạn (V và E là tập hữu hạn) mà thôi, chính vì vậy từ đây về sau, nếu không chú thích gì thêm thì khi nói tới đồ thị, ta hiểu rằng đó là đồ thị hữu hạn.

Cạnh liên thuộc, đỉnh kề, bậc

Đối với đồ thị vô hướng G = (V, E). Xét một cạnh $e \in E$, nếu e = (u, v) thì ta nói hai đỉnh u và v là **kề nhau** (adjacent) và cạnh e này **liên thuộc** (incident) với đỉnh u và đỉnh v.

Với một đỉnh v trong đồ thị, ta định nghĩa **bậc** (degree) của v, ký hiệu deg(v) là số cạnh liên thuộc với v. Dễ thấy rằng trên đơn đồ thị thì số cạnh liên thuộc với v cũng là số đỉnh kề với v.

Định lý: Giả sử G = (V, E) là đồ thị vô hướng với m cạnh, khi đó tổng tất cả các bậc đỉnh trong V sẽ bằng 2m:

$$\sum_{v \in V} \deg(v) = 2m$$

Chứng minh: Khi lấy tổng tất cả các bậc đỉnh tức là mỗi cạnh e = (u, v) bất kỳ sẽ được tính một lần trong deg(u) và một lần trong deg(v). Từ đó suy ra kết quả.

Hệ quả: Trong đồ thị vô hướng, số đỉnh bậc lẻ là số chẵn

Đối với đồ thị có hướng G = (V, E). Xét một cung $e \in E$, nếu e = (u, v) thì ta nói **u nối tới v** và **v nối từ u,** cung e là đi **ra khỏi đỉnh u và đi vào đỉnh v**. Đỉnh u khi đó được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của cung e.

Với mỗi đỉnh v trong đồ thị có hướng, ta định nghĩa: **Bán bậc ra** của v ký hiệu deg+(v) là số cung đi ra khỏi nó; **bán bậc vào** ký hiệu deg-(v) là số cung đi vào đỉnh đó

Định lý: Giả sử G = (V, E) là đồ thị có hướng với m cung, khi đó tổng tất cả các bán bậc ra của các đỉnh bằng tổng tất cả các bán bậc vào và bằng m:

$$\sum_{v\in V} \deg^-(v) = \sum_{v\in V} \deg^+(v) = m$$

Chứng minh: Khi lấy tổng tất cả các bán bậc ra hay bán bậc vào, mỗi cung (u, v) bất kỳ sẽ được tính đúng 1 lần trong $\deg^+(u)$ và cũng được tính đúng 1 lần trong $\deg^-(v)$. Từ đó suy ra kết quả

Một số tính chất của đồ thị có hướng không phụ thuộc vào hướng của các cung. Do đó để tiện trình bày, trong một số trường hợp ta có thể không quan tâm đến hướng của các cung và coi các cung đó là các cạnh của đồ thị vô hướng. Và đồ thị vô hướng đó được gọi là **đồ thị vô hướng nền** của đồ thị có hướng ban đầu.

§2. BIỂU DIỄN ĐỒ THỊ TRÊN MÁY TÍNH

2.1. MA TRẬN LIỀN KỀ (MA TRẬN KỀ)

Giả sử G = (V, E) là một **đơn đồ thị** có số đỉnh (ký hiệu $|V|$) là n, Không mất tính tổng quát có thể coi các đỉnh được đánh số 1, 2, …, n. Khi đó ta có thể biểu diễn đồ thị bằng một ma trận vuông $A = [a_{ij}]$ cấp n. Trong đó:

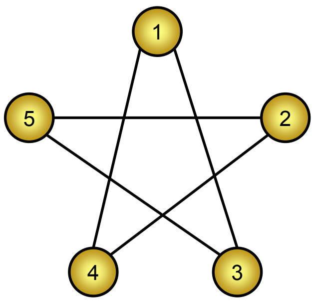
$a_{ij} = 1$ nếu $(i, j) \in E$

$a_{ij} = 0$ nếu $(i, j) \notin E$

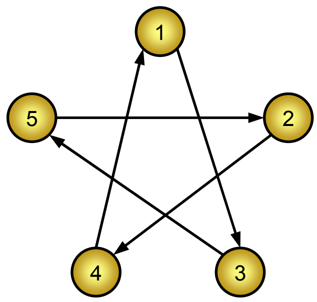
Quy ước $a_{ii} = 0$ với $\forall i$;

Đối với đa đồ thị thì việc biểu diễn cũng tương tự trên, chỉ có điều nếu như (i, j) là cạnh thì không phải ta ghi số 1 vào vị trí a_{ij} mà là ghi số cạnh nối giữa đỉnh i và đỉnh j.

Ví dụ:



$$A=\begin{bmatrix}0&0&1&1&0\\0&0&0&1&1\\1&0&0&0&1\\1&1&0&0&0\\0&1&1&0&0\end{bmatrix}$$



$$A=\begin{bmatrix}0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\1&0&0&0&0\\0&1&0&0&0\end{bmatrix}$$

Các tính chất của ma trận kề:

Đối với đồ thị vô hướng G, thì ma trận kề tương ứng là ma trận đối xứng ($a_{ij} = a_{ji}$), điều này không đúng với đồ thị có hướng.

Nếu G là đồ thị vô hướng và A là ma trận kề tương ứng thì trên ma trận A:

Tổng các số trên hàng i = Tổng các số trên cột i = Bậc của đỉnh i = deg(i)

Nếu G là đồ thị có hướng và A là ma trận kề tương ứng thì trên ma trận A:

Tổng các số trên hàng i = Bán bậc ra của đỉnh i = $deg^+(i)$

Tổng các số trên cột i = Bán bậc vào của đỉnh i = $deg^-(i)$

Trong trường hợp G là đơn đồ thị, ta có thể biểu diễn ma trận kề A tương ứng là các phần tử logic.

a_{ij} = TRUE nếu $(i, j) \in E$ và a_{ij} = FALSE nếu $(i, j) \notin E$

Ưu điểm của ma trận kề:

Đơn giản, trực quan, dễ cài đặt trên máy tính

Để kiểm tra xem hai đỉnh (u, v) của đồ thị có kề nhau hay không, ta chỉ việc kiểm tra bằng một phép so sánh: $a_{uv} \neq 0$.

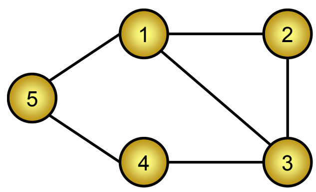
Nhược điểm của ma trận kề:

Bất kể số cạnh của đồ thị là nhiều hay ít, ma trận kề luôn luôn đòi hỏi n^2 ô nhớ để lưu các phần tử ma trận, điều đó gây lãng phí bộ nhớ dẫn tới việc không thể biểu diễn được đồ thị với số đỉnh lớn.

Với một đỉnh u bất kỳ của đồ thị, nhiều khi ta phải xét tất cả các đỉnh v khác kề với nó, hoặc xét tất cả các cạnh liên thuộc với nó. Trên ma trận kề việc đó được thực hiện bằng cách xét tất cả các đỉnh v và kiểm tra điều kiện $a_{uv} \neq 0$. Như vậy, ngay cả khi đỉnh u là **đỉnh cô lập** (không kề với đỉnh nào) hoặc **đỉnh treo** (chỉ kề với 1 đỉnh) ta cũng buộc phải xét tất cả các đỉnh và kiểm tra điều kiện trên dẫn tới lãng phí thời gian

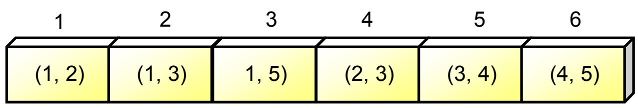
2.2. DANH SÁCH CẠNH

Trong trường hợp đồ thị có n đỉnh, m cạnh, ta có thể biểu diễn đồ thị dưới dạng danh sách cạnh bằng cách liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp (u, v) tương ứng với một cạnh của đồ thị. (Trong trường hợp đồ thị có hướng thì mỗi cặp (u, v) tương ứng với một cung, u là đỉnh đầu và v là đỉnh cuối của cung). Danh sách được lưu trong bộ nhớ dưới dạng mảng hoặc danh sách móc nối. Ví dụ với đồ thị ở Hình 53:

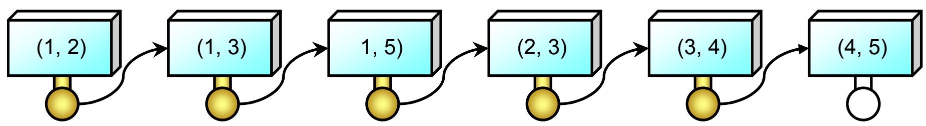


Hình 53

Cài đặt trên mảng:



Cài đặt trên danh sách móc nối:



Ưu điểm của danh sách cạnh:

Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ: chẳng hạn m < 6n), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần 2m ô nhớ để lưu danh sách cạnh.

Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các cạnh dễ dàng hơn. (Thuật toán Kruskal chẳng hạn)

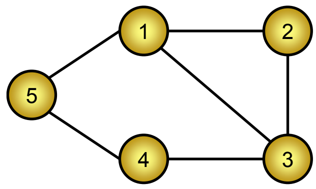
Nhược điểm của danh sách cạnh:

Nhược điểm cơ bản của danh sách cạnh là khi ta cần duyệt tất cả các đỉnh kề với đỉnh v nào đó của đồ thị, thì chẳng có cách nào khác là phải duyệt tất cả các cạnh, lọc ra những cạnh có chứa đỉnh v và xét đỉnh còn lại. Điều đó khá tốn thời gian trong trường hợp đồ thị dày (nhiều cạnh).

2.3. DANH SÁCH KỀ

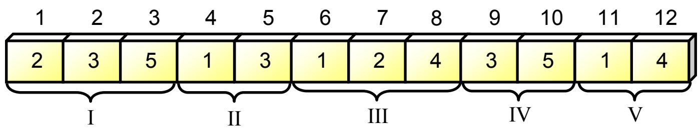
Để khắc phục nhược điểm của các phương pháp ma trận kề và danh sách cạnh, người ta đề xuất phương pháp biểu diễn đồ thị bằng danh sách kề. Trong cách biểu diễn này, với mỗi đỉnh v của đồ thị, ta cho tương ứng với nó một danh sách các đỉnh kề với v.

Với đồ thị G = (V, E). V gồm n đỉnh và E gồm m cạnh. Có hai cách cài đặt danh sách kề phổ biến:



Hình 54

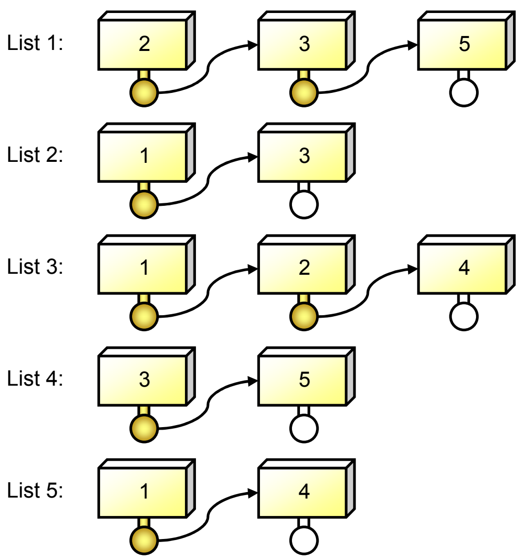
Cách 1: Dùng một mảng các đỉnh, mảng đó chia làm n đoạn, đoạn thứ i trong mảng lưu danh sách các đỉnh kề với đỉnh i: Với đồ thị ở Hình 54, danh sách kề sẽ là một mảng A gồm 12 phần tử:



Để biết một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng Head lưu vị trí riêng. Head[i] sẽ bằng chỉ số đứng liền trước đoạn thứ i. Quy ước Head[n + 1] bằng m. Với đồ thị bên thì mảng Head[1..6] sẽ là: (0, 3, 5, 8, 10, 12)

Trong mảng A, đoạn từ vị trí Head[i] + 1 đến Head[i + 1] sẽ chứa các đỉnh kề với đỉnh i. Lưu ý rằng với đồ thị có hướng gồm m cung thì cấu trúc này cần phải đủ chứa m phần tử, với đồ thị vô hướng m cạnh thì cấu trúc này cần phải đủ chứa 2m phần tử

Cách 2: Dùng các danh sách móc nối: Với mỗi đỉnh i của đồ thị, ta cho tương ứng với nó một danh sách móc nối các đỉnh kề với i, có nghĩa là tương ứng với một đỉnh i, ta phải lưu lại List[i] là chốt của một danh sách móc nối. Ví dụ với đồ thị ở Hình 54, các danh sách móc nối sẽ là:



Ưu điểm của danh sách kề:

Đối với danh sách kề, việc duyệt tất cả các đỉnh kề với một đỉnh v cho trước là hết sức dễ dàng, cái tên "danh sách kề" đã cho thấy rõ điều này. Việc duyệt tất cả các cạnh cũng đơn giản vì một cạnh thực ra là nối một đỉnh với một đỉnh khác kề nó.

Nhược điểm của danh sách kề

Danh sách kề yếu hơn ma trận kề ở việc kiểm tra (u, v) có phải là cạnh hay không, bởi trong cách biểu diễn này ta sẽ phải việc phải duyệt toàn bộ danh sách kề của u hay danh sách kề của v. Tuy nhiên đối với những thuật toán mà ta sẽ khảo sát, danh sách kề tốt hơn hẳn so với hai phương pháp biểu diễn trước. Chỉ có điều, trong trường hợp cụ thể mà ma trận kề hay danh sách cạnh **không thể hiện nhược điểm** thì ta nên dùng ma trận kề (hay danh sách cạnh) bởi cài đặt danh sách kề có phần dài dòng hơn.

2.4. NHẬN XÉT

Trên đây là nêu các cách biểu diễn đồ thị trong bộ nhớ của máy tính, còn nhập dữ liệu cho đồ thị thì có nhiều cách khác nhau, dùng cách nào thì tuỳ. Chẳng hạn nếu biểu diễn bằng ma trận kề mà cho nhập dữ liệu cả ma trận cấp n x n (n là số đỉnh) thì khi nhập từ bàn phím sẽ rất mất thời gian, ta cho nhập kiểu danh sách cạnh cho nhanh. Chẳng hạn mảng A (nxn) là ma trận kề của một đồ thị vô hướng thì ta có thể khởi tạo ban đầu mảng A gồm toàn số 0, sau đó cho người sử dụng nhập các cạnh bằng cách nhập các cặp (i, j); chương trình sẽ tăng A[i, j] và A[j, i] lên 1. Việc nhập có thể cho kết thúc khi người sử dụng nhập giá trị i = 0. Ví dụ:

```
program Nhap_Do_Thi;
var
  A: array[1..100, 1..100] of Integer; {Ma trận kề của đồ thị}
  n, i, j: Integer;
begin
  Write('Number of vertices'); ReadLn(n);
  FillChar(A, SizeOf(A), 0);
  repeat
    Write('Enter edge (i, j) (i = 0 to exit) ');
    ReadLn(i, j); {Nhập một cặp (i, j) tưởng như là nhập danh sách cạnh}
    if i <> 0 then
      begin  {nhưng lưu trữ trong bộ nhớ lại theo kiểu ma trận kề}
        Inc(A[i, j]);
        Inc(A[j, i]);
      end;
  until i = 0; {Nếu người sử dụng nhập giá trị i = 0 thì dừng quá trình nhập, nếu không thì tiếp tục}
end.
```

Trong nhiều trường hợp đủ không gian lưu trữ, việc chuyển đổi từ cách biểu diễn nào đó sang cách biểu diễn khác không có gì khó khăn. Nhưng đối với thuật toán này thì làm trên ma trận kề ngắn gọn hơn, đối với thuật toán kia có thể làm trên danh sách cạnh dễ dàng hơn v.v… Do đó, với mục đích dễ hiểu, các chương trình sau này sẽ lựa chọn phương pháp biểu diễn sao cho việc cài đặt đơn giản nhất nhằm nêu bật được bản chất thuật toán. Còn trong trường hợp cụ thể bắt buộc phải dùng một cách biểu diễn nào đó khác, thì việc sửa đổi chương trình cũng không tốn quá nhiều thời gian.

§3. CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

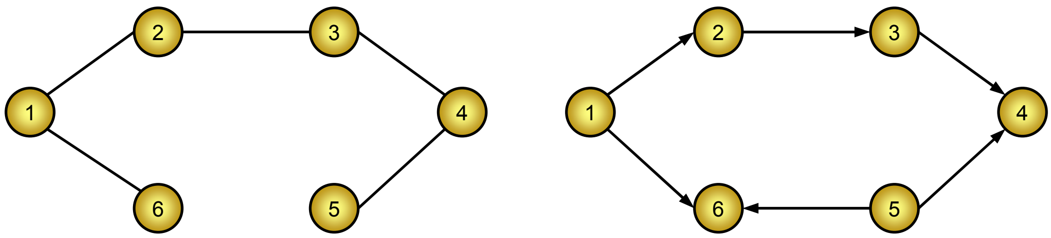
3.1. BÀI TOÁN

Cho đồ thị G = (V, E). u và v là hai đỉnh của G. Một **đường đi** (path) độ dài l từ đỉnh u đến đỉnh v là dãy $(u = x_0, x_1, \ldots, x_l = v)$ thoả mãn $(x_i, x_{i+1}) \in E$ với $\forall i: (0 \le i < l)$.

Đường đi nói trên còn có thể biểu diễn bởi dãy các cạnh: $(u = x_0, x_1), (x_1, x_2), \ldots, (x_{l-1}, x_l = v)$

Đỉnh u được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối gọi là **chu trình** (Circuit), đường đi không có cạnh nào đi qua hơn 1 lần gọi là **đường đi đơn,** tương tự ta có khái niệm **chu trình đơn.**

Ví dụ: Xét một đồ thị vô hướng và một đồ thị có hướng trong Hình 55:



Hình 55: Đồ thị và đường đi

Trên cả hai đồ thị, (1, 2, 3, 4) là đường đi đơn độ dài 3 từ đỉnh 1 tới đỉnh 4. (1, 6, 5, 4) không phải đường đi vì không có cạnh (cung) nối từ đỉnh 6 tới đỉnh 5.

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó. Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kỳ đỉnh nào. Chính vì vậy mà ta phải xây dựng những thuật toán cho phép **duyệt một cách hệ thống** các đỉnh, những thuật toán như vậy gọi là những thuật toán **tìm kiếm trên đồ thị** và ở đây ta quan tâm đến hai thuật toán cơ bản nhất: **thuật toán tìm kiếm theo chiều sâu** và **thuật toán tìm kiếm theo chiều rộng** cùng với một số ứng dụng của chúng.

Lưu ý:

Những cài đặt dưới đây là cho đơn đồ thị vô hướng, muốn làm với đồ thị có hướng hay đa đồ thị cũng không phải sửa đổi gì nhiều.

Dữ liệu về đồ thị sẽ được nhập từ file văn bản GRAPH.INP. Trong đó:

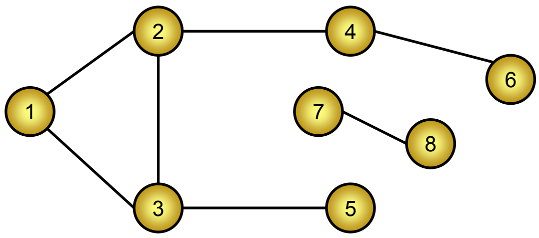
Dòng 1 chứa số đỉnh n (≤ 100), số cạnh m của đồ thị, đỉnh xuất phát S, đỉnh kết thúc F cách nhau một dấu cách.

m dòng tiếp theo, mỗi dòng có dạng hai số nguyên dương u, v cách nhau một dấu cách, thể hiện có cạnh nối đỉnh u và đỉnh v trong đồ thị.

Kết quả ghi ra file văn bản PATH.OUT

Danh sách các đỉnh có thể đến được từ S

Đường đi từ S tới F



GRAPH.INP	PATH.OUT
8 7 1 5 1 2 1 3 2 3 2 4 3 5 4 6 7 8	From 1 you can visit: 1, 2, 3, 5, 4, 6, Path from 1 to 5: 5<-3<-2<-1

3.2. THUẬT TOÁN TÌM KIẾM THEO CHIỀU SÂU (DEPTH FIRST SEARCH)

3.2.1. Cài đặt đệ quy

Tư tưởng của thuật toán có thể trình bày như sau: Trước hết, mọi đỉnh x kề với S tất nhiên sẽ đến được từ S. Với mỗi đỉnh x kề với S đó thì tất nhiên những đỉnh y kề với x cũng đến được từ S… Điều đó gợi ý cho ta viết một thủ tục đệ quy DFS(u) mô tả việc duyệt từ đỉnh u bằng cách thông báo thăm đỉnh u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh chưa thăm kề với u.

Để không một đỉnh nào bị liệt kê tới hai lần, ta sử dụng kỹ thuật đánh dấu, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa

Để lưu lại đường đi từ đỉnh xuất phát S, trong thủ tục DFS(u), trước khi gọi đệ quy DFS(v) với v là một đỉnh kề với u mà chưa đánh dấu, ta lưu lại vết đường đi từ u tới v bằng cách đặt TRACE[v] := u, tức là TRACE[v] lưu lại đỉnh liền trước v trong đường đi từ S tới v. Khi quá trình tìm kiếm theo chiều sâu kết thúc, đường đi từ S tới F sẽ là:

$$F \leftarrow p_1 = \text{Trace}[F] \leftarrow p_2 = \text{Trace}[p_1] \leftarrow \ldots \leftarrow S.$$

```
procedure DFS(u∈V);
begin
  < 1. Thông báo tới được u >;
  < 2. Đánh dấu u là đã thăm (có thể tới được từ S)>;
  < 3. Xét mọi đỉnh v kề với u mà chưa thăm, với mỗi đỉnh v đó >;
    begin
      Trace[v] := u; {Lưu vết đường đi, đỉnh mà từ đó tới v là u}
      DFS(v); {Gọi đệ quy duyệt tương tự đối với v}
    end;
end;

begin {Chương trình chính}
  < Nhập dữ liệu: đồ thị, đỉnh xuất phát S, đỉnh đích F >;
  < Khởi tạo: Tất cả các đỉnh đều chưa bị đánh dấu >;
  DFS(S);
  < Nếu F chưa bị đánh dấu thì không thể có đường đi từ S tới F >;
  < Nếu F đã bị đánh dấu thì truy theo vết để tìm đường đi từ S tới F >;
end.
```

P_4_03_1.PAS * Thuật toán tìm kiếm theo chiều sâu

```
program Depth_First_Search_1;
const
  InputFile  = 'GRAPH.INP';
```

```
  OutputFile = 'PATH.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị}
  Free: array[1..max] of Boolean; {Free[v] = True ⇔ v chưa được thăm đến}
  Trace: array[1..max] of Integer; {Trace[v] = đỉnh liền trước v trên đường đi từ S tới v}
  n, S, F: Integer;
  fo: Text;

procedure Enter; {Nhập dữ liệu}
var
  i, u, v, m: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(a, SizeOf(a), False); {Khởi tạo đồ thị chưa có cạnh nào}
  ReadLn(fi, n, m, S, F); {Đọc dòng 1 ra 4 số n, m, S và F}
  for i := 1 to m do {Đọc m dòng tiếp ra danh sách cạnh}
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(fi);
end;

procedure DFS(u: Integer); {Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh u}
var
  v: Integer;
begin
  Write(fo, u, ', '); {Thông báo tới được u}
  Free[u] := False; {Đánh dấu u đã thăm}
  for v := 1 to n do
    if Free[v] and a[u, v] then {Với mỗi đỉnh v chưa thăm kề với u}
      begin
        Trace[v] := u; {Lưu vết đường đi: Đỉnh liền trước v trong đường đi từ S tới v là u}
        DFS(v); {Tiếp tục tìm kiếm theo chiều sâu bắt đầu từ v}
      end;
end;

procedure Result; {In đường đi từ S tới F}
begin
  WriteLn(fo); {Vào dòng thứ hai của Output file}
  WriteLn(fo, 'Path from ', S, ' to ', F, ': ');
  if Free[F] then {Nếu F chưa đánh dấu thăm tức là không có đường}
    WriteLn(fo,'not found')
  else {Truy vết đường đi, bắt đầu từ F}
    begin
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
end;

begin
  Enter;
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, 'From ', S, ' you can visit: ');
  FillChar(Free, n, True);
  DFS(S);
  Result;
```

```
  Close(fo);
end.
```

Chú ý:

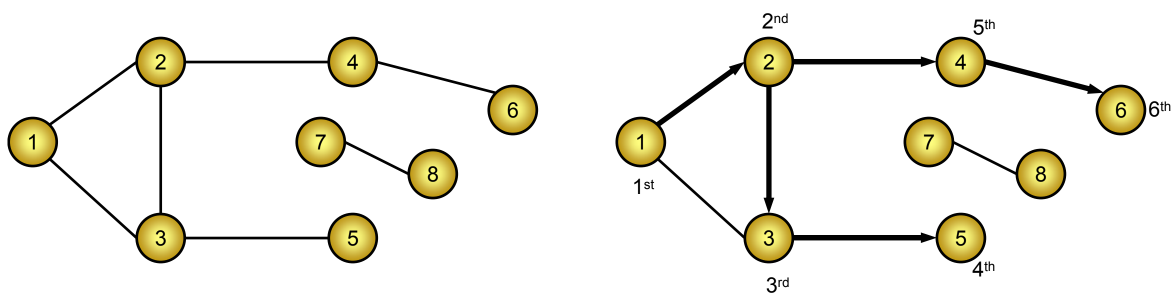
Vì có kỹ thuật đánh dấu, nên thủ tục DFS sẽ được gọi ≤ n lần (n là số đỉnh)

Đường đi từ S tới F có thể có nhiều, ở trên chỉ là một trong số các đường đi. Cụ thể là đường đi có thứ tự từ điển nhỏ nhất.

Có thể chẳng cần dùng mảng đánh dấu Free, ta khởi tạo mảng lưu vết Trace ban đầu toàn 0, mỗi lần từ đỉnh u thăm đỉnh v, ta có thao tác gán vết Trace[v] := u, khi đó Trace[v] sẽ khác 0. Vậy việc kiểm tra một đỉnh v là chưa được thăm ta có thể kiểm tra Trace[v] = 0. Chú ý: ban đầu khởi tạo Trace[S] := -1 (Chỉ là để cho khác 0 thôi).

```
procedure DFS(u: Integer); {Cải tiến}
var
  v: Integer;
begin
  Write(u, ', ');
  for v := 1 to n do
    if (Trace[v] = 0) and A[u, v] then {Trace[v] = 0 thay vì Free[v] = True}
      begin
        Trace[v] := u; {Lưu vết cũng là đánh dấu luôn}
        DFS(v);
      end;
end;
```

Ví dụ: Với đồ thị sau đây, đỉnh xuất phát S = 1: quá trình duyệt đệ quy có thể vẽ trên cây tìm kiếm DFS sau (Mũi tên u→v chỉ thao tác đệ quy: DFS(u) gọi DFS(v)).



Hình 56: Cây DFS

Hỏi: Đỉnh 2 và 3 đều kề với đỉnh 1, nhưng tại sao DFS(1) chỉ gọi đệ quy tới DFS(2) mà không gọi DFS(3) ?.

Trả lời: Đúng là cả 2 và 3 đều kề với 1, nhưng DFS(1) sẽ tìm thấy 2 trước và gọi DFS(2). Trong DFS(2) sẽ xét tất cả các đỉnh kề với 2 mà chưa đánh dấu thì dĩ nhiên trước hết nó tìm thấy 3 và gọi DFS(3), khi đó 3 đã bị đánh dấu nên khi kết thúc quá trình đệ quy gọi DFS(2), lùi về DFS(1) thì đỉnh 3 đã được thăm (đã bị đánh dấu) nên DFS(1) sẽ không gọi DFS(3) nữa.

Hỏi: Nếu F = 5 thì đường đi từ 1 tới 5 trong chương trình trên sẽ in ra thế nào ?.

Trả lời: DFS(5) do DFS(3) gọi nên Trace[5] = 3. DFS(3) do DFS(2) gọi nên Trace[3] = 2. DFS(2) do DFS(1) gọi nên Trace[2] = 1. Vậy đường đi là: 5 ← 3 ← 2 ←1.

Với cây thể hiện quá trình đệ quy DFS ở trên, ta thấy nếu dây chuyền đệ quy là: DFS(S) → DFS (u_1) → DFS(u_2) … Thì thủ tục DFS nào gọi cuối dây chuyền sẽ được thoát ra đầu tiên, thủ tục DFS(S)

gọi đầu dây chuyền sẽ được thoát cuối cùng, từ đây ta có ý tưởng mô phỏng dây chuyền đệ quy bằng một ngăn xếp (Stack).

3.2.2. Cài đặt không đệ quy

Khi mô tả quá trình đệ quy bằng một ngăn xếp, ta luôn luôn để cho ngăn xếp lưu lại dây chuyền duyệt sâu từ nút gốc (đỉnh xuất phát S).

```
<Thăm S, đánh dấu S đã thăm>;
<Đẩy S vào ngăn xếp>; {Dây chuyền đệ quy ban đầu chỉ có một đỉnh S}
repeat
  <Lấy u khỏi ngăn xếp>; {Đang đứng ở đỉnh u}
  if <u có đỉnh kề chưa thăm> then
    begin
      <Chỉ chọn lấy 1 đỉnh v, là đỉnh đầu tiên kề u mà chưa được thăm>;
      <Thông báo thăm v>;
      <Đẩy u trở lại ngăn xếp>; {Giữ lại địa chỉ quay lui}
      <Đẩy tiếp v vào ngăn xếp>; {Dây chuyền duyệt sâu được "nối" thêm v nữa}
    end;
  {Còn nếu u không có đỉnh kề chưa thăm thì ngăn xếp sẽ ngắn lại, tương ứng với sự lùi về của dây chuyền DFS}
until <Ngăn xếp rỗng>;
```

P_4_03_2.PAS * Thuật toán tìm kiếm theo chiều sâu không đệ quy

```
program Depth_First_Search_2;
const
  InputFile  = 'GRAPH.INP';
  OutputFile = 'PATH.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Free: array[1..max] of Boolean;
  Trace: array[1..max] of Integer;
  Stack: array[1..max] of Integer;
  n, S, F, Last: Integer;
  fo: Text;

procedure Enter;
var
  i, u, v, m: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(a, SizeOf(a), False);
  ReadLn(fi, n, m, S, F);
  for i := 1 to m do
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(fi);
end;

procedure Init; {Khởi tạo}
begin
  FillChar(Free, n, True); {Các đỉnh đều chưa đánh dấu}
  Last := 0; {Ngăn xếp rỗng}
end;

procedure Push(V: Integer); {Đẩy một đỉnh V vào ngăn xếp}
begin
  Inc(Last);
```

```
  Stack[Last] := V;
end;

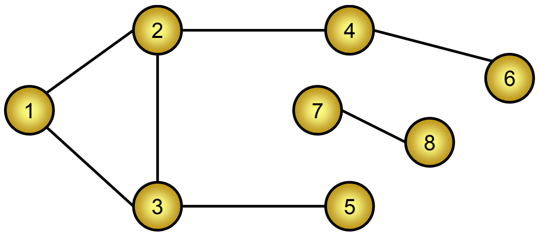
function Pop: Integer; {Lấy một đỉnh khỏi ngăn xếp, trả về trong kết quả hàm}
begin
  Pop := Stack[Last];
  Dec(Last);
end;

procedure DFS;
var
  u, v: Integer;
begin
  Write(fo, S, ', '); Free[S] := False; {Thăm S, đánh dấu S đã thăm}
  Push(S); {Khởi động dây chuyền duyệt sâu}
  repeat
 {Dây chuyền duyệt sâu đang là S→ …→ u}
    u := Pop; {u là điểm cuối của dây chuyền duyệt sâu hiện tại}
    for v := 1 to n do
      if Free[v] and a[u, v] then {Chọn v là đỉnh đầu tiên chưa thăm kề với u, nếu có:}
        begin
          Write(fo, v, ', '); Free[v] := False; {Thăm v, đánh dấu v đã thăm}
          Trace[v] := u; {Lưu vết đường đi}
          Push(u); Push(v); {Dây chuyền duyệt sâu bây giờ là S→ …→ u→ v}
          Break;
        end;
  until Last = 0; {Ngăn xếp rỗng}
end;

procedure Result; {In đường đi từ S tới F}
begin
  WriteLn(fo); {Vào dòng thứ hai của Output file}
  WriteLn(fo, 'Path from ', S, ' to ', F, ': ');
  if Free[F] then {Nếu F chưa đánh dấu thăm tức là không có đường}
    WriteLn(fo,'not found')
  else {Truy vết đường đi, bắt đầu từ F}
    begin
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
end;

begin
  Enter;
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, 'From ', S, ' you can visit: ');
  Init;
  DFS;
  Result;
  Close(fo);
end.
```

Ví dụ: Với đồ thị dưới đây (S = 1), Ta thử theo dõi quá trình thực hiện thủ tục tìm kiếm theo chiều sâu dùng ngăn xếp và đối sánh thứ tự các đỉnh được thăm với thứ tự từ 1st đến 6th trong cây tìm kiếm của thủ tục DFS dùng đệ quy.



Trước hết ta thăm đỉnh 1 và đẩy nó vào ngăn xếp.

Bước lặp	Ngăn xếp	u	v	Ngăn xếp sau mỗi bước	Giải thích
1	(1)	1	**2**	(1, 2)	Tiến sâu xuống thăm 2
2	(1, 2)	2	**3**	(1, 2, 3)	Tiến sâu xuống thăm 3
3	(1, 2, 3)	3	**5**	(1, 2, 3, 5)	Tiến sâu xuống thăm 5
4	(1, 2, 3, 5)	5	**Không có**	(1, 2, 3)	Lùi lại
5	(1, 2, 3)	3	**Không có**	(1, 2)	Lùi lại
6	(1, 2)	2	**4**	(1, 2, 4)	Tiến sâu xuống thăm 4
7	(1, 2, 4)	4	**6**	(1, 2, 4, 6)	Tiến sâu xuống thăm 6
8	(1, 2, 4, 6)	6	**Không có**	(1, 2, 4)	Lùi lại
9	(1, 2, 4)	4	**Không có**	(1, 2)	Lùi lại
10	(1, 2)	2	**Không có**	(1)	Lùi lại
11	(1)	1	**Không có**	∅	Lùi hết dây chuyền, Xong

Trên đây là phương pháp dựa vào tính chất của thủ tục đệ quy để tìm ra phương pháp mô phỏng nó. Tuy nhiên, trên mô hình đồ thị thì ta có thể có một cách viết khác tốt hơn cũng không đệ quy: Thử nhìn lại cách thăm đỉnh của DFS: Từ một đỉnh u, chọn lấy một đỉnh v kề nó mà chưa thăm rồi tiến sâu xuống thăm v. Còn nếu mọi đỉnh kề u đều đã thăm thì lùi lại một bước và lặp lại quá trình tương tự, việc lùi lại này có thể thực hiện dễ dàng mà không cần dùng Stack nào cả, bởi với mỗi đỉnh u đã có một nhãn Trace[u] (là đỉnh mà đã từ đó mà ta tới thăm u), khi quay lui từ u sẽ lùi về đó.

Vậy nếu ta đang đứng ở đỉnh u, thì đỉnh kế tiếp phải thăm tới sẽ được tìm như trong hàm FindNext dưới đây:

```
function FindNext(u∈V): ∈V; {Tìm đỉnh sẽ thăm sau đỉnh u, trả về 0 nếu mọi đỉnh tới được từ S đều đã thăm}
begin
  repeat
    for (∀v ∈ Kề(u)) do
      if <v chưa thăm> then    {Nếu u có đỉnh kề chưa thăm thì chọn đỉnh kề đầu tiên chưa thăm để thăm tiếp}
        begin
          Trace[v] := u; {Lưu vết}
          FindNext := v;
          Exit;
        end;
    u := Trace[u]; {Nếu không, lùi về một bước. Lưu ý là Trace[S] được gán bằng n + 1}
  until u = n + 1;
  FindNext := 0; {ở trên không Exit được tức là mọi đỉnh tới được từ S đã duyệt xong}
end;

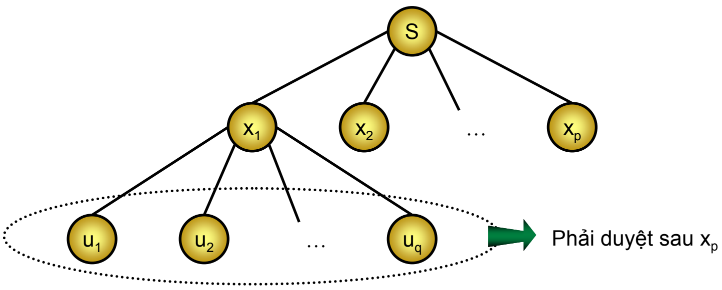
begin {Thuật toán duyệt theo chiều sâu}
  Trace[S] := n + 1;
  <Khởi tạo các đỉnh đều là chưa thăm>
  u := S;
  repeat
    <Thông báo thăm u, đánh dấu u đã thăm>;
```

```
    u := FindNext(u);
  until u = 0;
end;
```

3.3. THUẬT TOÁN TÌM KIẾM THEO CHIỀU RỘNG (BREADTH FIRST SEARCH)

3.3.1. Cài đặt bằng hàng đợi

Cơ sở của phương pháp cài đặt này là "lập lịch" duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh kề nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần S hơn sẽ được duyệt trước). Ví dụ: Bắt đầu ta thăm đỉnh S. Việc thăm đỉnh S sẽ phát sinh thứ tự duyệt những đỉnh (x_1, x_2, …, x_p) kề với S (những đỉnh gần S nhất). Khi thăm đỉnh x_1 sẽ lại phát sinh yêu cầu duyệt những đỉnh (u_1, u_2 …, u_q) kề với x_1. Nhưng rõ ràng các đỉnh u này "xa" S hơn những đỉnh x nên chúng chỉ được duyệt khi tất cả những đỉnh x đã duyệt xong. Tức là thứ tự duyệt đỉnh sau khi đã thăm x_1 sẽ là: (x_2, x_3…, x_p, u_1, u_2, …, u_q).



Hình 57: Cây BFS

Giả sử ta có một danh sách chứa những đỉnh đang "chờ" thăm. Tại mỗi bước, ta thăm một đỉnh đầu danh sách và cho những đỉnh chưa "xếp hàng" kề với nó xếp hàng thêm vào cuối danh sách. Chính vì nguyên tắc đó nên danh sách chứa những đỉnh đang chờ sẽ được tổ chức dưới dạng hàng đợi (Queue)

Mô hình của giải thuật có thể viết như sau:

Bước 1: Khởi tạo:

Các đỉnh đều ở trạng thái chưa đánh dấu, ngoại trừ đỉnh xuất phát S là đã đánh dấu

Một hàng đợi (Queue), ban đầu chỉ có một phần tử là S. Hàng đợi dùng để chứa các đỉnh sẽ được duyệt theo thứ tự ưu tiên chiều rộng

Bước 2: Lặp các bước sau đến khi hàng đợi rỗng:

Lấy u khỏi hàng đợi, thông báo thăm u (Bắt đầu việc duyệt đỉnh u)

Xét tất cả những đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:

Đánh dấu v.

Ghi nhận vết đường đi từ u tới v (Có thể làm chung với việc đánh dấu)

Đẩy v vào hàng đợi (v sẽ chờ được duyệt tại những bước sau)

Bước 3: Truy vết tìm đường đi.

P_4_03_3.PAS * Thuật toán tìm kiếm theo chiều rộng dùng hàng đợi

```
program Breadth_First_Search_1;
const
  InputFile  = 'GRAPH.INP';
  OutputFile = 'PATH.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Free: array[1..max] of Boolean; {Free[v] ⇔ v chưa được xếp vào hàng đợi để chờ thăm}
  Trace: array[1..max] of Integer;
  Queue: array[1..max] of Integer;
  n, S, F, First, Last: Integer;
  fo: Text;

procedure Enter; {Nhập dữ liệu}
var
  i, u, v, m: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(a, SizeOf(a), False);
  ReadLn(fi, n, m, S, F);
  for i := 1 to m do
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(fi);
end;

procedure Init; {Khởi tạo}
begin
  FillChar(Free, n, True); {Các đỉnh đều chưa đánh dấu}
  Free[S] := False; {Ngoại trừ đỉnh S}
  Queue[1] := S; {Hàng đợi chỉ gồm có một đỉnh S}
  Last := 1;
  First := 1;
end;

procedure Push(V: Integer); {Đẩy một đỉnh V vào hàng đợi}
begin
  Inc(Last);
  Queue[Last] := V;
end;

function Pop: Integer; {Lấy một đỉnh khỏi hàng đợi, trả về trong kết quả hàm}
begin
  Pop := Queue[First];
  Inc(First);
end;

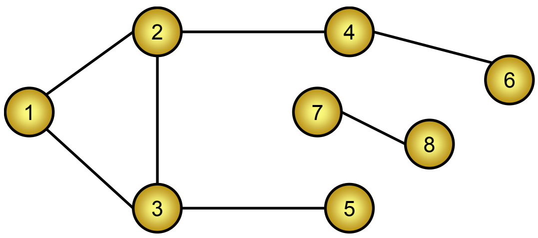
procedure BFS; {Thuật toán tìm kiếm theo chiều rộng}
var
  u, v: Integer;
begin
  repeat
    u := Pop; {Lấy một đỉnh u khỏi hàng đợi}
    Write(fo, u, ', '); {Thông báo thăm u}
    for v := 1 to n do
      if Free[v] and a[u, v] then {Xét những đỉnh v chưa đánh dấu kề u}
```

```
      begin
        Push(v); {Đưa v vào hàng đợi để chờ thăm}
        Free[v] := False; {Đánh dấu v}
        Trace[v] := u; {Lưu vết đường đi: đỉnh liền trước v trong đường đi từ S là u}
      end;
  until First > Last; {Cho tới khi hàng đợi rỗng}
end;

procedure Result; {In đường đi từ S tới F}
begin
  WriteLn(fo); {Vào dòng thứ hai của Output file}
  WriteLn(fo, 'Path from ', S, ' to ', F, ': ');
  if Free[F] then {Nếu F chưa đánh dấu thăm tức là không có đường}
    WriteLn(fo,'not found')
  else {Truy vết đường đi, bắt đầu từ F}
    begin
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
end;

begin
  Enter;
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, 'From ', S, ' you can visit: ');
  Init;
  BFS;
  Result;
  Close(fo);
end.
```

Ví dụ: Xét đồ thị dưới đây, Đỉnh xuất phát S = 1.



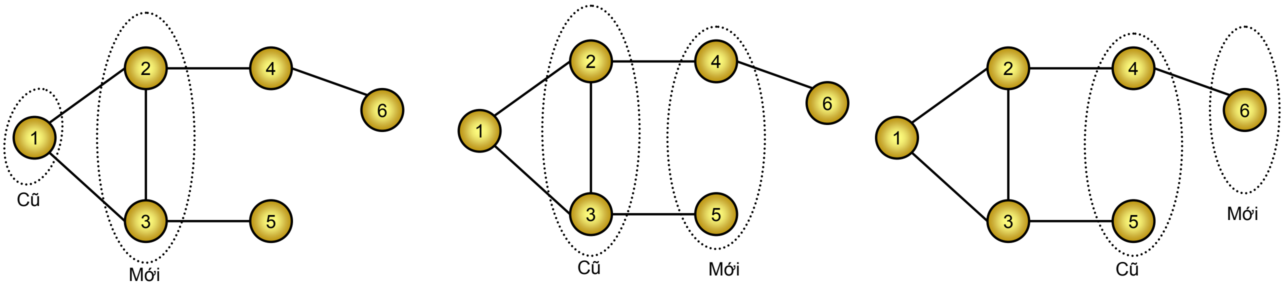
Hàng đợi	**Đỉnh u (lấy ra từ hàng đợi)**	Hàng đợi (sau khi lấy u ra)	Các đỉnh v kề u mà chưa lên lịch	Hàng đợi sau khi đẩy những đỉnh v vào
(1)	**1**	∅	2, 3	(2, 3)
(2, 3)	**2**	(3)	4	(3, 4)
(3, 4)	**3**	(4)	5	(4, 5)
(4, 5)	**4**	(5)	6	(5, 6)
(5, 6)	**5**	(6)	Không có	(6)
(6)	**6**	∅	Không có	∅

Để ý thứ tự các phần tử lấy ra khỏi hàng đợi, ta thấy trước hết là 1; sau đó đến 2, 3; rồi mới tới 4, 5; cuối cùng là 6. Rõ ràng là đỉnh gần S hơn sẽ được duyệt trước. Và như vậy, ta có nhận xét: nếu kết

hợp lưu vết tìm đường đi thì **đường đi từ S tới F sẽ là đường đi ngắn nhất** (theo nghĩa qua ít cạnh nhất)

3.3.2. Cài đặt bằng thuật toán loang

Cách cài đặt này sử dụng hai tập hợp, một tập "cũ" chứa những đỉnh "đang xét", một tập "mới" chứa những đỉnh "sẽ xét". Ban đầu tập "cũ" chỉ gồm mỗi đỉnh xuất phát, tại mỗi bước ta sẽ dùng tập "cũ" tính tập "mới", tập "mới" sẽ gồm những đỉnh chưa được thăm mà kề với một đỉnh nào đó của tập "cũ". Lặp lại công việc trên (sau khi đã gán tập "cũ" bằng tập "mới") cho tới khi tập cũ là rỗng:



Hình 58: Thuật toán loang

Giải thuật loang có thể dựng như sau:

Bước 1: Khởi tạo

Các đỉnh khác S đều chưa bị đánh dấu, đỉnh S bị đánh dấu, tập "cũ" Old := {S}

Bước 2: Lặp các bước sau đến khi Old = ∅

Đặt tập "mới" New = ∅, sau đó dùng tập "cũ" tính tập "mới" như sau:

Xét các đỉnh u ∈ Old, với mỗi đỉnh u đó:

Thông báo thăm u

Xét tất cả những đỉnh v kề với u mà chưa bị đánh dấu, với mỗi đỉnh v đó:

Đánh dấu v

Lưu vết đường đi, đỉnh liền trước v trong đường đi S→v là u

Đưa v vào tập New

Gán tập "cũ" Old := tập "mới" New và lặp lại (có thể luân phiên vai trò hai tập này)

Bước 3: Truy vết tìm đường đi.

P_4_03_4.PAS * Thuật toán tìm kiếm theo chiều rộng dùng phương pháp loang

```
program Breadth_First_Search_2;
const
  InputFile  = 'GRAPH.INP';
  OutputFile = 'PATH.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Free: array[1..max] of Boolean;
  Trace: array[1..max] of Integer;
  Old, New: set of Byte;
  n, S, F: Byte;
```

```
  fo: Text;

procedure Enter; {Nhập dữ liệu}
var
  i, u, v, m: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(a, SizeOf(a), False);
  ReadLn(fi, n, m, S, F);
  for i := 1 to m do
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(fi);
end;

procedure Init;
begin
  FillChar(Free, n, True);
  Free[S] := False; {Các đỉnh đều chưa đánh dấu, ngoại trừ đỉnh S đã đánh dấu}
  Old := [S]; {Tập "cũ" khởi tạo ban đầu chỉ có mỗi S}
end;

procedure BFS; {Thuật toán loang}
var
  u, v: Byte;
begin
  repeat {Lặp: dùng Old tính New}
    New := [];
    for u := 1 to n do
      if u in Old then {Xét những đỉnh u trong tập Old, với mỗi đỉnh u đó:}
        begin
          Write(fo, u, ', '); {Thông báo thăm u}
          for v := 1 to n do
            if Free[v] and a[u, v] then {Quét tất cả những đỉnh v chưa bị đánh dấu mà kề với u}
              begin
                Free[v] := False; {Đánh dấu v và lưu vết đường đi}
                Trace[v] := u;
                New := New + [v]; {Đưa v vào tập New}
              end;
        end;
    Old := New; {Gán tập "cũ" := tập "mới" và lặp lại}
  until Old = []; {Cho tới khi không loang được nữa}
end;

procedure Result; {In đường đi từ S tới F}
begin
  WriteLn(fo); {Vào dòng thứ hai của Output file}
  WriteLn(fo, 'Path from ', S, ' to ', F, ': ');
  if Free[F] then {Nếu F chưa đánh dấu thăm tức là không có đường}
    WriteLn(fo,'not found')
  else {Truy vết đường đi, bắt đầu từ F}
    begin
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
end;
```

```
begin
  Enter;
  Assign(fo, OutputFile); Rewrite(fo);
  WriteLn(fo, 'From ', S, ' you can visit: ');
  Init;
  BFS;
  Result;
  Close(fo);
end.
```

3.4. ĐỘ PHỨC TẠP TÍNH TOÁN CỦA BFS VÀ DFS

Quá trình tìm kiếm trên đồ thị bắt đầu từ một đỉnh có thể thăm tất cả các đỉnh còn lại, khi đó cách biểu diễn đồ thị có ảnh hưởng lớn tới chi phí về thời gian thực hiện giải thuật:

Trong trường hợp ta biểu diễn đồ thị bằng danh sách kề, cả hai thuật toán BFS và DFS đều có độ phức tạp tính toán là O(n + m) = O(max(n, m)). Đây là cách cài đặt tốt nhất.

Nếu ta biểu diễn đồ thị bằng ma trận kề như ở trên thì độ phức tạp tính toán trong trường hợp này là $O(n + n^2) = O(n^2)$.

Nếu ta biểu diễn đồ thị bằng danh sách cạnh, thao tác duyệt những đỉnh kề với đỉnh u sẽ dẫn tới việc phải duyệt qua toàn bộ danh sách cạnh, đây là cài đặt tồi nhất, nó có độ phức tạp tính toán là O(n.m).

Bài tập

Mê cung hình chữ nhật kích thước m x n gồm các ô vuông đơn vị. Trên mỗi ô ký tự:

O: Nếu ô đó an toàn

X: Nếu ô đó có cạm bẫy

E: Nếu là ô có một nhà thám hiểm đang đứng.

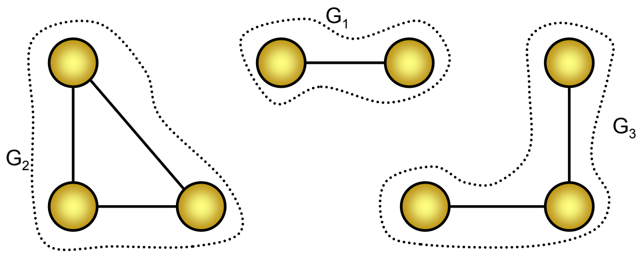
Duy nhất chỉ có 1 ô ghi chữ E. Nhà thám hiểm có thể từ một ô đi sang một trong số các ô chung cạnh với ô đang đứng. Một cách đi thoát khỏi mê cung là một hành trình đi qua các ô an toàn ra một ô biên. Hãy chỉ giúp cho nhà thám hiểm một hành trình thoát ra khỏi mê cung

§4. TÍNH LIÊN THÔNG CỦA ĐỒ THỊ

4.1. ĐỊNH NGHĨA

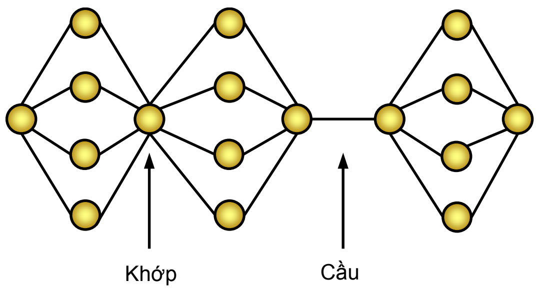
4.1.1. Đối với đồ thị vô hướng G = (V, E)

G gọi là **liên thông** (connected) nếu luôn tồn tại đường đi giữa mọi cặp đỉnh phân biệt của đồ thị. Nếu G không liên thông thì chắc chắn nó sẽ là hợp của hai hay nhiều đồ thị con* liên thông, các đồ thị con này đôi một không có đỉnh chung. Các đồ thị con liên thông rời nhau như vậy được gọi là các thành phần liên thông của đồ thị đang xét (Xem ví dụ).



Hình 59: Đồ thị G và các thành phần liên thông G1, G2, G3 của nó

Đôi khi, việc xoá đi một đỉnh và tất cả các cạnh liên thuộc với nó sẽ tạo ra một đồ thị con mới có nhiều thành phần liên thông hơn đồ thị ban đầu, các đỉnh như thế gọi là **đỉnh cắt** hay **điểm khớp**. Hoàn toàn tương tự, những cạnh mà khi ta bỏ nó đi sẽ tạo ra một đồ thị có nhiều thành phần liên thông hơn so với đồ thị ban đầu được gọi là một **cạnh cắt** hay một **cầu.**



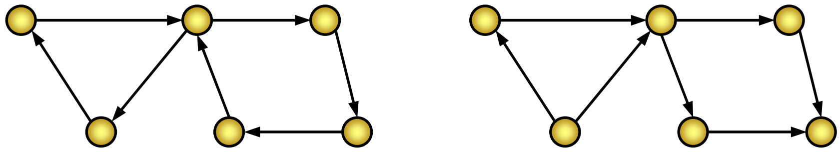
Hình 60: Khớp và cầu

4.1.2. Đối với đồ thị có hướng G = (V, E)

Có hai khái niệm về tính liên thông của đồ thị có hướng tuỳ theo chúng ta có quan tâm tới hướng của các cung không.

* Đồ thị G = (V, E) là con của đồ thị G' = (V', E') nếu G là đồ thị có $V \subseteq V'$ và $E \subseteq E'$

G gọi là **liên thông mạnh** (Strongly connected) nếu luôn tồn tại đường đi (theo các cung định hướng) giữa hai đỉnh bất kỳ của đồ thị, g gọi là **liên thông yếu** (weakly connected) nếu đồ thị vô hướng nền của nó là liên thông



Hình 61: Liên thông mạnh và liên thông yếu

4.2. TÍNH LIÊN THÔNG TRONG ĐỒ THỊ VÔ HƯỚNG

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán kiểm tra tính liên thông của đồ thị vô hướng hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông của đồ thị vô hướng.

Giả sử đồ thị vô hướng G = (V, E) có n đỉnh đánh số 1, 2, …, n.

Để liệt kê các thành phần liên thông của G phương pháp cơ bản nhất là:

Đánh dấu đỉnh 1 và những đỉnh có thể đến từ 1, thông báo những đỉnh đó thuộc thành phần liên thông thứ nhất.

Nếu tất cả các đỉnh đều đã bị đánh dấu thì G là đồ thị liên thông, nếu không thì sẽ tồn tại một đỉnh v nào đó chưa bị đánh dấu, ta sẽ đánh dấu v và các đỉnh có thể đến được từ v, thông báo những đỉnh đó thuộc thành phần liên thông thứ hai.

Và cứ tiếp tục như vậy cho tới khi tất cả các đỉnh đều đã bị đánh dấu

```
procedure Duyệt(u)
begin
  <Dùng BFS hoặc DFS liệt kê và đánh dấu những đỉnh có thể đến được từ u>
end;
begin
  for ∀ v ∈ V do <khởi tạo v chưa đánh dấu>;
  Count := 0;
  for u := 1 to n do
    if <u chưa đánh dấu> then
      begin
        Count := Count + 1;
        WriteLn('Thành phần liên thông thứ ', Count, ' gồm các đỉnh : ');
        Duyệt(u);
      end;
end.
```

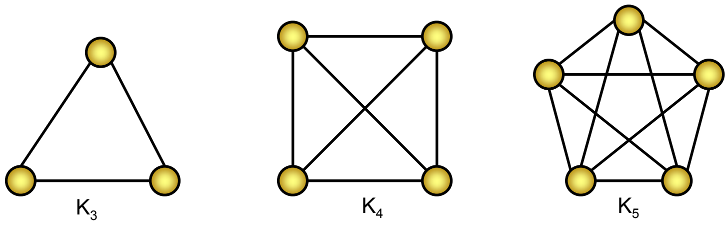
Với thuật toán liệt kê các thành phần liên thông như thế này, thì độ phức tạp tính toán của nó đúng bằng độ phức tạp tính toán của thuật toán tìm kiếm trên đồ thị trong thủ tục Duyệt.

4.3. ĐỒ THỊ ĐẦY ĐỦ VÀ THUẬT TOÁN WARSHALL

4.3.1. Định nghĩa:

Đồ thị đầy đủ với n đỉnh, ký hiệu K_n, là một đơn đồ thị vô hướng mà giữa hai đỉnh bất kỳ của nó đều có cạnh nối.

Đồ thị đầy đủ K_n có đúng: $C_n^2 = \frac{n.(n-1)}{2}$ cạnh và bậc của mọi đỉnh đều bằng n - 1.



Hình 62: Đồ thị đầy đủ

4.3.2. Bao đóng đồ thị:

Với đồ thị G = (V, E), người ta xây dựng đồ thị G' = (V, E') cũng gồm những đỉnh của G còn các cạnh xây dựng như sau: (ở đây quy ước giữa u và u luôn có đường đi)

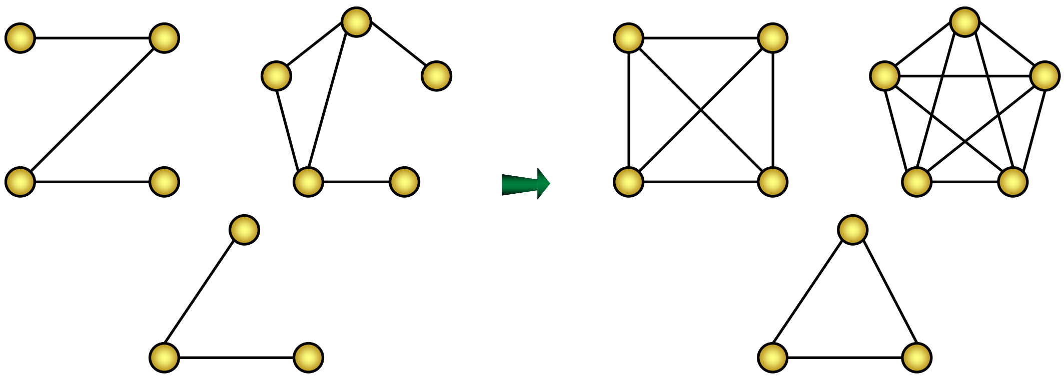
Giữa đỉnh u và v của G' có cạnh nối $\Leftrightarrow$ Giữa đỉnh u và v của G có đường đi

Đồ thị G' xây dựng như vậy được gọi là bao đóng của đồ thị G.

Từ định nghĩa của đồ thị đầy đủ, ta dễ dàng suy ra một đồ thị đầy đủ bao giờ cũng liên thông và từ định nghĩa đồ thị liên thông, ta cũng dễ dàng suy ra được:

Một đơn đồ thị vô hướnglà liên thông nếu và chỉ nếu bao đóng của nó là đồ thị đầy đủ

Một đơn đồ thị vô hướng có k thành phần liên thông nếu và chỉ nếu bao đóng của nó có k thành phần liên thông đầy đủ.



Hình 63: Đơn đồ thị vô hướng và bao đóng của nó

Bởi việc kiểm tra một đồ thị có phải đồ thị đầy đủ hay không có thể thực hiện khá dễ dàng (đếm số cạnh chẳng hạn) nên người ta nảy ra ý tưởng có thể kiểm tra tính liên thông của đồ thị thông qua việc kiểm tra tính đầy đủ của bao đóng. Vấn đề đặt ra là phải có thuật toán xây dựng bao đóng của một đồ thị cho trước và một trong những thuật toán đó là:

4.3.3. Thuật toán Warshall

Thuật toán Warshall - gọi theo tên của Stephen Warshall, người đã mô tả thuật toán này vào năm 1960, đôi khi còn được gọi là thuật toán Roy-Warshall vì Roy cũng đã mô tả thuật toán này vào năm 1959. Thuật toán đó có thể mô tả rất gọn:

Từ ma trận kề A của đơn đồ thị vô hướng G (a_{ij} = True nếu (i, j) là cạnh của G) ta sẽ sửa đổi A để nó trở thành ma trận kề của bao đóng bằng cách: **Với mọi đỉnh k xét theo thứ tự từ 1 tới n, ta xét tất cả các cặp đỉnh (u, v): nếu có cạnh nối (u, k) (a_{uk} = True) và có cạnh nối (k, v) (a_{kv} = True) thì ta tự nối thêm cạnh (u, v) nếu nó chưa có (đặt a_{uv} := True)**. Tư tưởng này dựa trên một quan sát đơn giản như sau: Nếu từ u có đường đi tới k và từ k lại có đường đi tới v thì tất nhiên từ u sẽ có đường đi tới v.

Với n là số đỉnh của đồ thị, ta có thể viết thuật toán Warshall như sau:

```
for k := 1 to n do
  for u := 1 to n do
    if a[u, k] then
      for v := 1 to n do
        if a[k, v] then a[u, v] := True;
```

hoặc

```
for k := 1 to n do
  for u := 1 to n do
    for v := 1 to n do
      a[u, v] := a[u, v] or a[u, k] and a[k, v];
```

Việc chứng minh tính đúng đắn của thuật toán đòi hỏi phải lật lại các lý thuyết về bao đóng bắc cầu và quan hệ liên thông, ta sẽ không trình bày ở đây. Có nhận xét rằng tuy thuật toán Warshall rất dễ cài đặt nhưng độ phức tạp tính toán của thuật toán này khá lớn ($O(n^3)$).

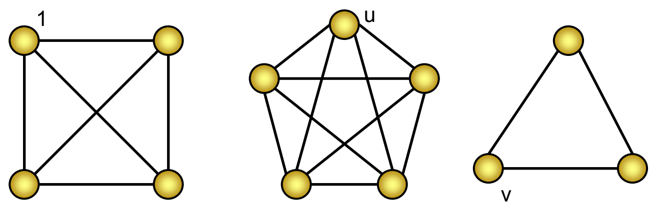
Dưới đây, ta sẽ thử cài đặt thuật toán Warshall tìm bao đóng của đơn đồ thị vô hướng sau đó đếm số thành phần liên thông của đồ thị:

Việc cài đặt thuật toán sẽ qua những bước sau:

Nhập ma trận kề A của đồ thị (Lưu ý ở đây A[v, v] luôn được coi là True với ∀v)

Dùng thuật toán Warshall tìm bao đóng, khi đó A là ma trận kề của bao đóng đồ thị

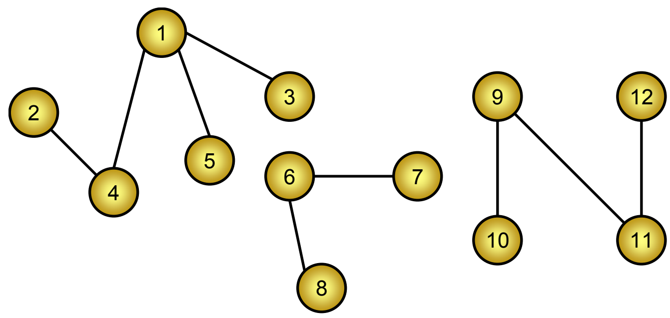
Dựa vào ma trận kề A, đỉnh 1 và những đỉnh kề với nó sẽ thuộc thành phần liên thông thứ nhất; với đỉnh u nào đó không kề với đỉnh 1, thì u cùng với những đỉnh kề nó sẽ thuộc thành phần liên thông thứ hai; với đỉnh v nào đó không kề với cả đỉnh 1 và đỉnh u, thì v cùng với những đỉnh kề nó sẽ thuộc thành phần liên thông thứ ba v.v…



Input: file văn bản GRAPH.INP

- Dòng 1: Chứa số đỉnh n (≤ 100) và số cạnh m của đồ thị cách nhau ít nhất một dấu cách
- m dòng tiếp theo, mỗi dòng chứa một cặp số u và v cách nhau ít nhất một dấu cách tượng trưng cho một cạnh (u, v)

Output: file văn bản CONNECT.OUT, liệt kê các thành phần liên thông



GRAPH.INP	CONNECT.OUT
12 9 1 3 1 4 1 5 2 4 6 7 6 8 9 10 9 11 11 12	Connected Component 1: 1, 2, 3, 4, 5, Connected Component 2: 6, 7, 8, Connected Component 3: 9, 10, 11, 12,

P_4_04_1.PAS * Thuật toán Warshall liệt kê các thành phần liên thông

```
program Connectivity;
const
  InputFile  = 'GRAPH.INP';
  OutputFile = 'CONNECT.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị}
  Free: array[1..max] of Boolean; {Free[v] = True ⇔ v chưa được liệt kê vào thành phần liên thông nào}
  k, u, v, n: Integer;
  Count: Integer;
  fo: Text;

procedure Enter; {Nhập đồ thị}
var
  i, u, v, m: Integer;
  fi: Text;
begin
  FillChar(a, SizeOf(a), False);
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m);
  for v := 1 to n do a[v, v] := True; {Dĩ nhiên từ v có đường đi đến chính v}
  for i := 1 to m do
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(fi);
end;

begin
  Enter;
  {Thuật toán Warshall}
  for k := 1 to n do
    for u := 1 to n do
      for v := 1 to n do
        a[u, v] := a[u, v] or a[u, k] and a[k, v];
  Assign(fo, OutputFile); Rewrite(fo);
  Count := 0;
  FillChar(Free, n, True); {Mọi đỉnh đều chưa được liệt kê vào thành phần liên thông nào}
  for u := 1 to n do
    if Free[u] then {Với một đỉnh u chưa được liệt kê vào thành phần liên thông nào}
      begin
        Inc(Count);
        WriteLn(fo, 'Connected Component ', Count, ': ');
        for v := 1 to n do
          if a[u, v] then {Xét những đỉnh kề u (trên bao đóng)}
            begin
```

```
            Write(fo, v, ', '); {Liệt kê đỉnh đó vào thành phần liên thông chứa u}
            Free[v] := False; {Liệt kê đỉnh nào đánh dấu đỉnh đó}
          end;
        WriteLn(fo);
      end;
  Close(fo);
end.
```

4.4. CÁC THÀNH PHẦN LIÊN THÔNG MẠNH

Đối với đồ thị có hướng, người ta quan tâm đến bài toán kiểm tra tính liên thông mạnh, hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông mạnh của đồ thị có hướng. Đối với bài toán đó ta có một phương pháp khá hữu hiệu dựa trên thuật toán tìm kiếm theo chiều sâu Depth First Search.

4.4.1. Phân tích

Thêm vào đồ thị một đỉnh x và nối x với tất cả các đỉnh còn lại của đồ thị bằng các cung định hướng. Khi đó quá trình tìm kiếm theo chiều sâu bắt đầu từ x có thể coi như một quá trình xây dựng cây tìm kiếm theo chiều sâu (cây DFS) gốc x.

```
procedure Visit(u∈V);
begin
  <Thêm u vào cây tìm kiếm DFS>;
  for (∀v: (u, v)∈E) do
    if <v không thuộc cây DFS> then Visit(v);
end;

begin
  <Thêm vào đồ thị đỉnh x và các cung định hướng (x, v) với mọi v>;
  <Khởi tạo cây tìm kiếm DFS := ∅>;
  Visit(x);
end.
```

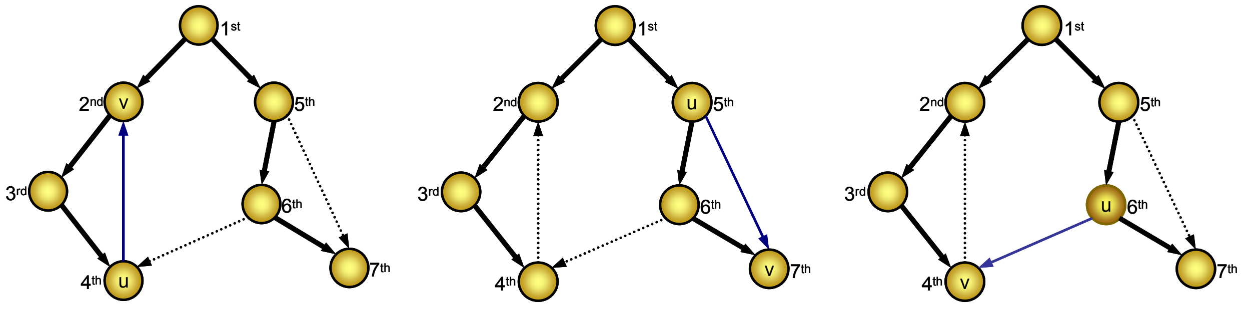
Để ý thủ tục thăm đỉnh đệ quy Visit(u). Thủ tục này xét tất cả những đỉnh v nối từ u, nếu v chưa được thăm thì đi theo cung đó thăm v, tức là bổ sung cung (u, v) vào cây tìm kiếm DFS. Nếu **v đã thăm** thì có ba khả năng xảy ra đối với vị trí của u và v trong cây tìm kiếm DFS:

v là tiền bối (ancestor - tổ tiên) của u, tức là v được thăm trước u và thủ tục Visit(u) do dây chuyền đệ quy từ thủ tục Visit(v) gọi tới. Cung (u, v) khi đó được gọi là **cung ngược** (Back edge)

v là hậu duệ (descendant - con cháu) của u, tức là u được thăm trước v, nhưng thủ tục Visit(u) sau khi tiến đệ quy theo một hướng khác đã gọi Visit(v) rồi. Nên khi dây chuyền đệ quy lùi lại về thủ tục Visit(u) sẽ thấy v là đã thăm nên không thăm lại nữa. Cung (u, v) khi đó gọi là **cung xuôi** (Forward edge).

v thuộc một nhánh của cây DFS đã duyệt trước đó, tức là sẽ có một đỉnh w được thăm trước cả u và v. Thủ tục Visit(w) gọi trước sẽ rẽ theo một nhánh nào đó thăm v trước, rồi khi lùi lại, rẽ sang một nhánh khác thăm u. Cung (u, v) khi đó gọi là **cung chéo** (Cross edge)

(Rất tiếc là từ điển thuật ngữ tin học Anh-Việt quá nghèo nàn nên không thể tìm ra những từ tương đương với các thuật ngữ ở trên. Ta có thể hiểu qua các ví dụ).



TH1: v là tiền bối của u
(u, v) là cung ngược

TH2: v là hậu duệ của u
(u, v) là cung xuôi

TH3: v nằm ở nhánh DFS đã duyệt trước u
(u, v là cung chéo)

Hình 64: Ba dạng cung ngoài cây DFS

Ta nhận thấy một đặc điểm của thuật toán tìm kiếm theo chiều sâu, thuật toán không chỉ duyệt qua các đỉnh, nó còn duyệt qua tất cả những cung nữa. Ngoài những cung nằm trên cây tìm kiếm, những cung còn lại có thể chia làm ba loại: cung ngược, cung xuôi, cung chéo.

4.4.2. Cây tìm kiếm DFS và các thành phần liên thông mạnh

Định lý 1:

Nếu a, b là hai đỉnh thuộc thành phần liên thông mạnh C thì với mọi đường đi từ a tới b cũng như từ b tới a. Tất cả đỉnh trung gian trên đường đi đó đều phải thuộc C.

Chứng minh

Nếu a và b là hai đỉnh thuộc C thì tức là có một đường đi từ a tới b và một đường đi khác từ b tới a. Suy ra với một đỉnh v nằm trên đường đi từ a tới b thì **a tới được v**, v tới được b, mà b có đường tới a nên **v cũng tới được a**. Vậy v nằm trong thành phần liên thông mạnh chứa a tức là v∈C. Tương tự với một đỉnh nằm trên đường đi từ b tới a.

Định lý 2:

Với một thành phần liên thông mạnh C bất kỳ, sẽ tồn tại một đỉnh r ∈C sao cho mọi đỉnh của C đều thuộc nhánh DFS gốc r.

Chứng minh:

Trước hết, nhắc lại một thành phần liên thông mạnh là một đồ thị con liên thông mạnh của đồ thị ban đầu thoả mãn tính chất tối đại tức là việc thêm vào thành phần đó một tập hợp đỉnh khác sẽ làm mất đi tính liên thông mạnh.

Trong số các đỉnh của C, chọn r là **đỉnh được thăm đầu tiên** theo thuật toán tìm kiếm theo chiều sâu. Ta sẽ chứng minh C nằm trong nhánh DFS gốc r. Thật vậy: với một đỉnh v bất kỳ của C, do C liên thông mạnh nên phải tồn tại một đường đi từ r tới v:

$$(r = x_0, x_1, \ldots, x_k = v)$$

Từ định lý 1, tất cả các đỉnh $x_1, x_2, \ldots, x_k$ đều thuộc C nên chúng sẽ phải thăm sau đỉnh r. Khi thủ tục Visit(r) được gọi thì tất cả các đỉnh $x_1, x_2 \ldots, x_k$=v đều chưa thăm; vì thủ tục Visit(r) sẽ liệt kê tất

cả những đỉnh chưa thăm đến được từ r bằng cách xây dựng nhánh gốc r của cây DFS, nên các đỉnh x_1, x_2, …, x_k = v sẽ thuộc nhánh gốc r của cây DFS. Bởi chọn v là đỉnh bất kỳ trong C nên ta có điều phải chứng minh.

Đỉnh r trong chứng minh định lý - **đỉnh thăm trước tất cả các đỉnh khác trong C** - gọi là **chốt** của thành phần C. Mỗi thành phần liên thông mạnh có duy nhất một chốt. Xét về vị trí trong cây tìm kiếm DFS, chốt của một thành phần liên thông là **đỉnh nằm cao nhất so với các đỉnh khác thuộc thành phần đó**, hay nói cách khác: **là tiền bối của tất cả các đỉnh thuộc thành phần đó**.

Định lý 3:

Luôn tìm được đỉnh chốt a thoả mãn: Quá trình tìm kiếm theo chiều sâu bắt đầu từ a không thăm được bất kỳ một chốt nào khác. (Tức là nhánh DFS gốc a không chứa một chốt nào ngoài a) chẳng hạn ta chọn a là chốt được thăm sau cùng trong một dây chuyền đệ quy hoặc chọn a là chốt thăm sau tất cả các chốt khác. Với chốt a như vậy thì các đỉnh thuộc **nhánh DFS gốc a chính là thành phần liên thông mạnh** chứa a.

Chứng minh:

Với mọi đỉnh v nằm trong nhánh DFS gốc a, xét b là chốt của thành phần liên thông mạnh chứa v. Ta sẽ chứng minh a ≡ b. Thật vậy, theo định lý 2, v phải nằm trong nhánh DFS gốc b. Vậy v nằm trong cả nhánh DFS gốc a và nhánh DFS gốc b. Giả sử phản chứng rằng a≠b thì sẽ có hai khả năng xảy ra:

Khả năng 1: Nhánh DFS gốc a chứa nhánh DFS gốc b, có nghĩa là thủ tục Visit(b) sẽ do thủ tục Visit(a) gọi tới, điều này mâu thuẫn với giả thiết rằng a là chốt mà quá trình tìm kiếm theo chiều sâu bắt đầu từ a không thăm một chốt nào khác.

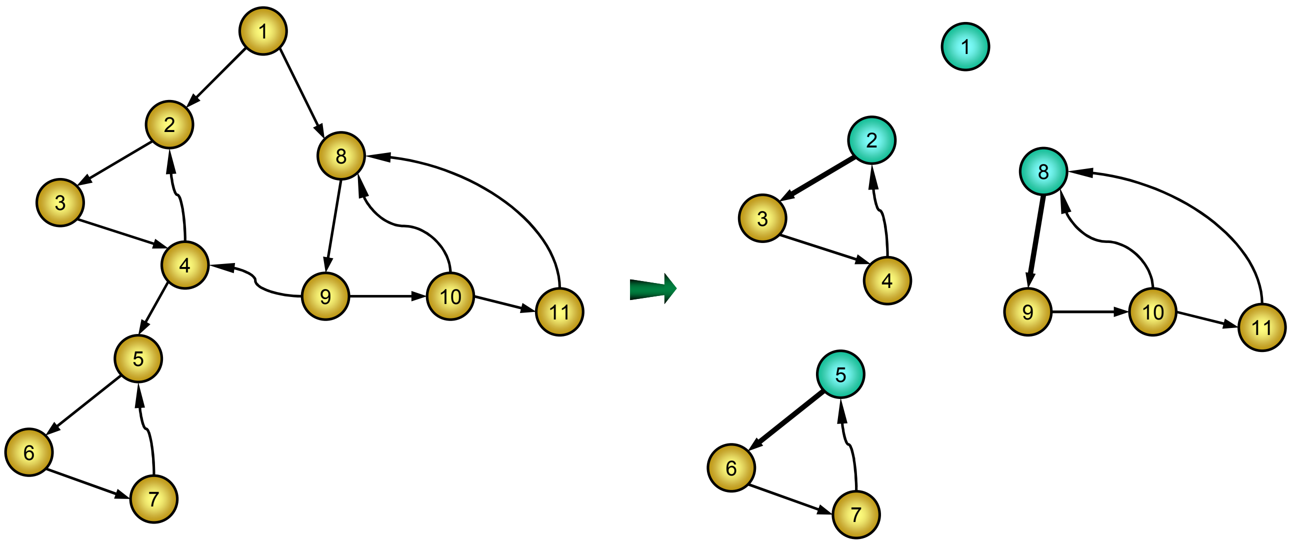
Khả năng 2: Nhánh DFS gốc a nằm trong nhánh DFS gốc b, có nghĩa là a nằm trên một đường đi từ b tới v. Do b và v thuộc cùng một thành phần liên thông mạnh nên theo định lý 1, a cũng phải thuộc thành phần liên thông mạnh đó. Vậy thì thành phần liên thông mạnh này có hai chốt a và b. Điều này vô lý.

Theo định lý 2, ta đã **có thành phần liên thông mạnh chứa a nằm trong nhánh DFS gốc a**, theo chứng minh trên ta lại có: Mọi đỉnh trong **nhánh DFS gốc a nằm trong thành phần liên thông mạnh chứa a**. Kết hợp lại được: Nhánh DFS gốc a chính là thành phần liên thông mạnh chứa a.

4.4.3. Thuật toán Tarjan (R.E.Tarjan - 1972)

Chọn u là chốt mà từ đó quá trình tìm kiếm theo chiều sâu không thăm thêm bất kỳ một chốt nào khác, chọn lấy thành phần liên thông mạnh thứ nhất là nhánh DFS gốc u. Sau đó loại bỏ nhánh DFS gốc u ra khỏi cây DFS, lại tìm thấy một đỉnh chốt v khác mà nhánh DFS gốc v không chứa chốt nào khác, lại chọn lấy thành phần liên thông mạnh thứ hai là nhánh DFS gốc v. Tương tự như vậy cho

thành phần liên thông mạnh thứ ba, thứ tư, v.v… Có thể hình dung thuật toán Tarjan "bẻ" cây DFS tại vị trí các chốt để được các nhánh rời rạc, mỗi nhánh là một thành phần liên thông mạnh.



Hình 65: Thuật toán Tarjan "bẻ" cây DFS

Trình bày dài dòng như vậy, nhưng điều quan trọng nhất bây giờ mới nói tới: **Làm thế nào kiểm tra một đỉnh v nào đó có phải là chốt hay không ?**

Hãy để ý nhánh DFS gốc ở đỉnh r nào đó.

Nhận xét 1:

Nếu như từ các đỉnh thuộc nhánh gốc r này không có cung ngược hay cung chéo nào đi ra khỏi nhánh đó thì r là chốt. Điều này dễ hiểu bởi như vậy có nghĩa là từ r, đi theo các cung của đồ thị thì chỉ đến được những đỉnh thuộc nhánh đó mà thôi. Vậy:

Thành phần liên thông mạnh chứa r $\subset$ Tập các đỉnh có thể đến từ r = Nhánh DFS gốc r

nên r là chốt.

Nhận xét 2:

Nếu từ một đỉnh v nào đó của nhánh DFS gốc r có một cung ngược tới một đỉnh w là tiền bối của r, thì r không là chốt. Thật vậy: do có chu trình (w→r→v→w) nên w, r, v thuộc cùng một thành phần liên thông mạnh. Mà w được thăm trước r, điều này mâu thuẫn với cách xác định chốt (Xem lại định lý 2)

Nhận xét 3:

Vấn đề phức tạp gặp phải ở đây là nếu từ một đỉnh v của nhánh DFS gốc r, có một cung chéo đi tới một nhánh khác. Ta sẽ thiết lập giải thuật liệt kê thành phần liên thông mạnh ngay trong thủ tục Visit(u), khi mà đỉnh u đã **duyệt xong**, tức là khi **các đỉnh khác của nhánh DFS gốc u đều đã thăm** và quá trình thăm đệ quy lùi lại về Visit(u). Nếu như u là chốt, ta thông báo nhánh DFS gốc u là thành phần liên thông mạnh chứa u và loại ngay các đỉnh thuộc thành phần đó khỏi đồ thị cũng như khỏi cây DFS. Có thể chứng minh được tính đúng đắn của phương pháp này, bởi nếu nhánh

DFS gốc u chứa một chốt u' khác thì u' phải duyệt xong trước u và cả nhánh DFS gốc u' đã bị loại bỏ rồi. Hơn nữa còn có thể chứng minh được rằng, khi thuật toán tiến hành như trên thì nếu như **từ một đỉnh v của một nhánh DFS gốc r có một cung chéo đi tới một nhánh khác thì r không là chốt**.

Để chứng tỏ điều này, ta dựa vào tính chất của cây DFS: cung chéo sẽ nối từ một nhánh tới nhánh thăm trước đó, chứ không bao giờ có cung chéo đi tới nhánh thăm sau. Giả sử có cung chéo (v, v') đi từ v ∈ nhánh DFS gốc r tới v' ∉ nhánh DFS gốc r, gọi r' là chốt của thành phần liên thông chứa v'. Theo tính chất trên, v' phải thăm trước r, suy ra **r' cũng phải thăm trước r**. Có hai khả năng xảy ra:

Nếu r' thuộc nhánh DFS đã duyệt trước r thì r' sẽ được duyệt xong trước khi thăm r, tức là khi thăm r và cả sau này khi thăm v thì nhánh DFS gốc r' đã bị huỷ, cung chéo (v, v') sẽ không được tính đến nữa.

Nếu r' là tiền bối của r thì ta có r' **đến được r**, v nằm trong nhánh DFS gốc r nên **r đến được v**, **v đến được v'** vì (v, v') là cung, v' **lại đến được r'** bởi r' là chốt của thành phần liên thông mạnh chứa v'. Ta thiết lập được chu trình (r'→r→v→v'→r'), suy ra r' và r thuộc cùng một thành phần liên thông mạnh, r' đã là chốt nên r không thể là chốt nữa.

Từ ba nhận xét và cách cài đặt chương trình như trong nhận xét 3, Ta có: Đỉnh r là chốt **nếu và chỉ nếu** không tồn tại cung ngược hoặc cung chéo nối một đỉnh thuộc nhánh DFS gốc r với một đỉnh ngoài nhánh đó, hay nói cách khác: **r là chốt nếu và chỉ nếu không tồn tại cung nối từ một đỉnh thuộc nhánh DFS gốc r tới một đỉnh thăm trước r**.

Dưới đây là một cài đặt hết sức thông minh, chỉ cần sửa đổi một chút thủ tục Visit ở trên là ta có ngay phương pháp này. Nội dung của nó là đánh số thứ tự các đỉnh từ đỉnh được thăm đầu tiên đến đỉnh thăm sau cùng. Định nghĩa Numbering[u] là số thứ tự của đỉnh u theo cách đánh số đó. Ta tính thêm Low[u] là giá trị Numbering nhỏ nhất trong các đỉnh có thể đến được từ một đỉnh v nào đó của nhánh DFS gốc u bằng một cung (với giả thiết rằng u có một cung giả nối với chính u).

Cụ thể cách cực tiểu hoá Low[u] như sau:

Trong thủ tục Visit(u), trước hết ta đánh số thứ tự thăm cho đỉnh u và khởi gán

$$Low[u] := Numbering[u] \text{ (u có cung tới chính u)}$$

Xét tất cả những đỉnh v nối từ u:

Nếu v đã thăm thì ta cực tiểu hoá Low[u] theo công thức:

$$Low[u]_{mới} := \min(Low[u]_{cũ}, Numbering[v]).$$

Nếu v chưa thăm thì ta gọi đệ quy đi thăm v, sau đó cực tiểu hoá Low[u] theo công thức:

$$Low[u]_{mới} := \min(Low[u]_{cũ}, Low[v])$$

Dễ dàng chứng minh được tính đúng đắn của công thức tính.

Khi duyệt xong một đỉnh u (chuẩn bị thoát khỏi thủ tục Visit(u). Ta so sánh Low[u] và Numbering[u]. Nếu như Low[u] = Numbering[u] thì u là chốt, bởi không có cung nối từ một đỉnh

thuộc nhánh DFS gốc u tới một đỉnh thăm trước u. Khi đó chỉ việc liệt kê các đỉnh thuộc thành phần liên thông mạnh chứa u là nhánh DFS gốc u.

Để công việc dễ dàng hơn nữa, ta định nghĩa một danh sách L được tổ chức dưới dạng ngăn xếp và dùng ngăn xếp này để lấy ra các đỉnh thuộc một nhánh nào đó. Khi thăm tới một đỉnh u, ta đẩy ngay đỉnh u đó vào ngăn xếp, thì khi duyệt xong đỉnh u, mọi đỉnh thuộc nhánh DFS gốc u sẽ được đẩy vào ngăn xếp L ngay sau u. Nếu u là chốt, ta chỉ việc lấy các đỉnh ra khỏi ngăn xếp L cho tới khi lấy tới đỉnh u là sẽ được nhánh DFS gốc u cũng chính là thành phần liên thông mạnh chứa u.

```
procedure Visit(u∈V);
begin
  Count := Count + 1; Numbering[u] := Count; {Trước hết đánh số u}
  Low[u] := Numbering[u];
  <Đưa u vào cây DFS>;
  <Đẩy u vào ngăn xếp L>;
  for (∀v: (u, v)∈E) do
    if <v đã thăm> then
      Low[u] := min(Low[u], Numbering[v])
    else
      begin
        Visit(v);
        Low[u] := min(Low[u], Low[v]);
      end;
  if Numbering[u] = Low[u] then {Nếu u là chốt}
    begin
      <Thông báo thành phần liên thông mạnh với chốt u gồm có các đỉnh:>;
      repeat
        <Lấy từ ngăn xếp L ra một đỉnh v>;
        <Output v>;
        <Xoá đỉnh v khỏi đồ thị>;
      until v = u;
    end;
end;

begin
  <Thêm vào đồ thị một đỉnh x và các cung (x, v) với mọi v>;
  <Khởi tạo một biến đếm Count := 0>;
  <Khởi tạo một ngăn xếp L := ∅>;
  <Khởi tạo cây tìm kiếm DFS := ∅>;
  Visit(x)
end.
```

Bởi thuật toán Tarjan chỉ là sửa đổi một chút thuật toán DFS, các thao tác vào/ra ngăn xếp được thực hiện không quá n lần. Vậy nên nếu đồ thị có n đỉnh và m cung thì độ phức tạp tính toán của thuật toán Tarjan vẫn là O(n + m) trong trường hợp biểu diễn đồ thị bằng danh sách kề, là $O(n^2)$ trong trường hợp biểu diễn bằng ma trận kề và là O(n.m) trong trường hợp biểu diễn bằng danh sách cạnh.

Mọi thứ đã sẵn sàng, dưới đây là toàn bộ chương trình. Trong chương trình này, ta sử dụng:

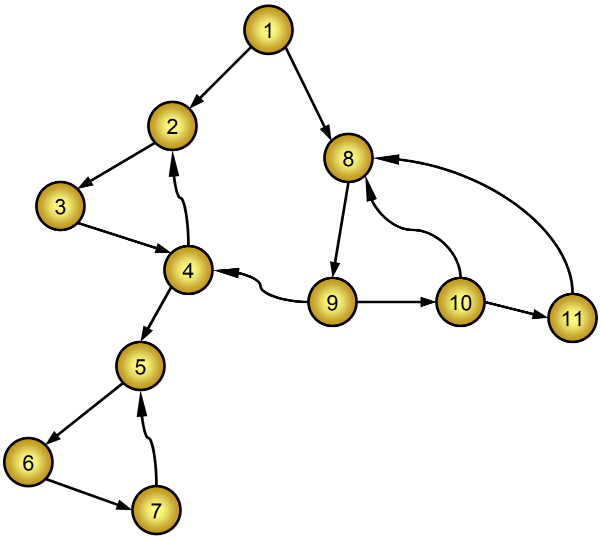
- Ma trận kề A để biểu diễn đồ thị.
- Mảng Free kiểu Boolean, Free[u] = True nếu u chưa bị liệt kê vào thành phần liên thông nào, tức là u chưa bị loại khỏi đồ thị.
- Mảng Numbering và Low với công dụng như trên, quy ước Numbering[u] = 0 nếu đỉnh u chưa được thăm.

- Mảng Stack, thủ tục Push, hàm Pop để mô tả cấu trúc ngăn xếp.

Input: file văn bản GRAPH.INP:

- Dòng đầu: Ghi số đỉnh n (≤ 100) và số cung m của đồ thị cách nhau một dấu cách
- m dòng tiếp theo, mỗi dòng ghi hai số nguyên u, v cách nhau một dấu cách thể hiện có cung (u, v) trong đồ thị

Output: file văn bản GRAPH.OUT, liệt kê các thành phần liên thông mạnh



GRAPH.INP	SCONNECT.OUT
11 15	Component 1:
1 2	7, 6, 5,
1 8	Component 2:
2 3	4, 3, 2,
3 4	Component 3:
4 2	11, 10, 9, 8,
4 5	Component 4:
5 6	1,
6 7	
7 5	
8 9	
9 4	
9 10	
10 8	
10 11	
11 8	

P_4_04_2.PAS * Thuật toán Tarjan liệt kê các thành phần liên thông mạnh

```
program Strong_connectivity;
const
  InputFile  = 'GRAPH.INP';
  OutputFile = 'GRAPH.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  Free: array[1..max] of Boolean;
  Numbering, Low, Stack: array[1..max] of Integer;
  n, Count, ComponentCount, Last: Integer;
  fo: Text;

procedure Enter;
var
  i, u, v, m: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(a, SizeOf(a), False);
  ReadLn(fi, n, m);
  for i := 1 to m do
    begin
      ReadLn(fi, u, v);
      a[u, v] := True;
    end;
  Close(fi);
end;

procedure Init; {Khởi tạo}
begin
  FillChar(Numbering, SizeOf(Numbering), 0); {Mọi đỉnh đều chưa thăm}
  FillChar(Free, SizeOf(Free), True); {Chưa đỉnh nào bị loại}
  Last := 0; {Ngăn xếp rỗng}
```

```
  Count := 0; {Biến đánh số thứ tự thăm}
  ComponentCount := 0; {Biến đánh số các thành phần liên thông}
end;

procedure Push(v: Integer); {Đẩy một đỉnh v vào ngăn xếp}
begin
  Inc(Last);
  Stack[Last] := v;
end;

function Pop: Integer; {Lấy một đỉnh khỏi ngăn xếp, trả về trong kết quả hàm}
begin
  Pop := Stack[Last];
  Dec(Last);
end;

function Min(x, y: Integer): Integer;
begin
  if x < y then Min := x else Min := y;
end;

procedure Visit(u: Integer); {Thuật toán tìm kiếm theo chiều sâu bắt đầu từ u}
var
  v: Integer;
begin
  Inc(Count); Numbering[u] := Count; {Trước hết đánh số cho u}
  Low[u] := Numbering[u]; {Coi u có cung tới u, nên có thể khởi gán Low[u] thế này rồi sau cực tiểu hoá dần}
  Push(u); {Đẩy u vào ngăn xếp}
  for v := 1 to n do
    if Free[v] and a[u, v] then {Xét những đỉnh v kề u}
      if Numbering[v] <> 0 then {Nếu v đã thăm}
        Low[u] := Min(Low[u], Numbering[v]) {Cực tiểu hoá Low[u] theo công thức này}
      else {Nếu v chưa thăm}
        begin
          Visit(v); {Tiếp tục tìm kiếm theo chiều sâu bắt đầu từ v}
          Low[u] := Min(Low[u], Low[v]); {Rồi cực tiểu hoá Low[u] theo công thức này}
        end;
  {Đến đây thì đỉnh u được duyệt xong, tức là các đỉnh thuộc nhánh DFS gốc u đều đã thăm}
  if Numbering[u] = Low[u] then {Nếu u là chốt}
    begin {Liệt kê thành phần liên thông mạnh có chốt u}
      Inc(ComponentCount);
      WriteLn(fo, 'Component ', ComponentCount, ': ');
      repeat
        v := Pop; {Lấy dần các đỉnh ra khỏi ngăn xếp}
        Write(fo, v, ', '); {Liệt kê các đỉnh đó}
        Free[v] := False; {Rồi loại luôn khỏi đồ thị}
      until v = u; {Cho tới khi lấy tới đỉnh u}
      WriteLn(fo);
    end;
end;

procedure Solve;
var
  u: Integer;
begin
  {Thay vì thêm một đỉnh giả x và các cung (x, v) với mọi đỉnh v rồi gọi Visit(x), ta có thể làm thế này cho nhanh}
  {sau này đỡ phải huỷ bỏ thành phần liên thông gồm mỗi một đỉnh giả đó}
  for u := 1 to n do
    if Numbering[u] = 0 then Visit(u);
end;

begin
  Enter;
  Assign(fo, OutputFile); Rewrite(fo);
```

```
  Init;
  Solve;
  Close(fo);
end.
```

Bài tập

Bài 1

Phương pháp cài đặt như trên có thể nói là rất hay và hiệu quả, đòi hỏi ta phải hiểu rõ bản chất thuật toán, nếu không thì rất dễ nhầm. Trên thực tế, còn có một phương pháp khác dễ hiểu hơn, tuy tính hiệu quả có kém hơn một chút. Hãy viết chương trình mô tả phương pháp sau:

Vẫn dùng thuật toán tìm kiếm theo chiều sâu với thủ tục Visit nói ở đầu mục, đánh số lại các đỉnh từ 1 tới n theo thứ tự ***duyệt xong****, sau đó đảo chiều tất cả các cung của đồ thị. Xét lần lượt các đỉnh theo thứ tự từ đỉnh duyệt xong sau cùng tới đỉnh duyệt xong đầu tiên, với mỗi đỉnh đó, ta lại dùng thuật toán tìm kiếm trên đồ thị (BFS hay DFS) liệt kê những đỉnh nào đến được từ đỉnh đang xét, đó chính là một thành phần liên thông mạnh. Lưu ý là khi liệt kê xong thành phần nào, ta loại ngay các đỉnh của thành phần đó khỏi đồ thị.*

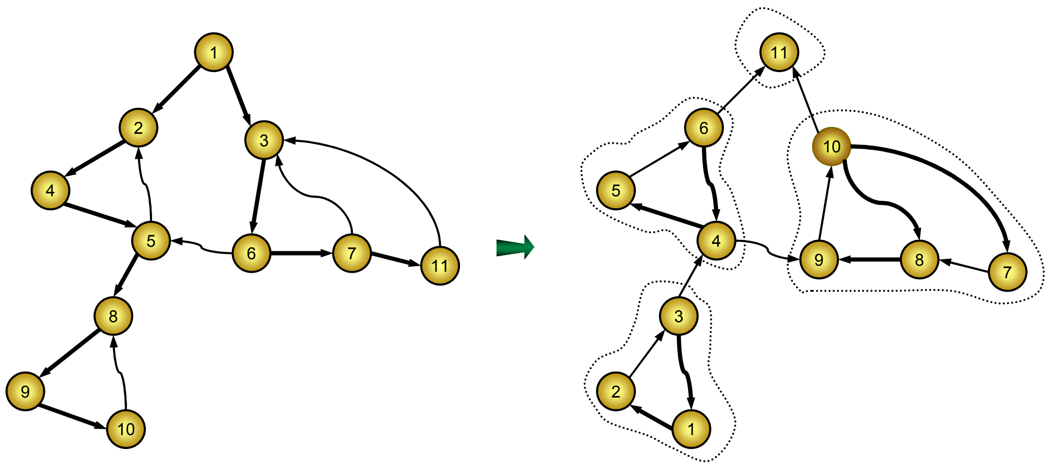
Tính đúng đắn của phương pháp có thể hình dung không mấy khó khăn:

Trước hết ta thêm vào đồ thị đỉnh x và các cung (x, v) với mọi v, sau đó gọi Visit(x) để xây dựng cây DFS gốc x. Hiển nhiên x là chốt của thành phần liên thông chỉ gồm mỗi x. Sau đó bỏ đỉnh x khỏi cây DFS, cây sẽ phân rã thành các cây con.

Đỉnh r duyệt xong sau cùng chắc chắn là gốc của một cây con (bởi khi duyệt xong nó chắc chắn sẽ lùi về x) suy ra r là chốt. Hơn thế nữa, nếu một đỉnh u nào đó tới được r thì u cũng phải thuộc cây con gốc r. Bởi nếu giả sử phản chứng rằng u thuộc cây con khác thì u phải được thăm trước r (do cây con gốc r được thăm tới sau cùng), có nghĩa là khi Visit(u) thì r chưa thăm. Vậy nên r sẽ thuộc nhánh DFS gốc u, mâu thuẫn với lập luận r là gốc. Từ đó suy ra nếu u tới được r thì r tới được u, tức là khi đảo chiều các cung, nếu r tới được đỉnh nào thì đỉnh đó thuộc thành phần liên thông chốt r.

Loại bỏ thành phần liên thông với chốt r khỏi đồ thị. Cây con gốc r lại phân rã thành nhiều cây con. Lập luận tương tự như trên với v' là đỉnh duyệt xong sau cùng (Hình 66).

Ví dụ:



Hình 66: Đánh số lại, đảo chiều các cung và duyệt BFS với cách chọn các đỉnh xuất phát ngược lại với thứ tự duyệt xong (thứ tự 11, 10… 3, 2, 1)

Bài 2

Thuật toán Warshall có thể áp dụng tìm bao đóng của đồ thị có hướng, vậy hãy kiểm tra tính liên thông mạnh của một đồ thị có hướng bằng hai cách: Dùng các thuật toán tìm kiếm trên đồ thị và thuật toán Warshall, sau đó so sánh ưu, nhược điểm của mỗi phương pháp

Bài 3

Trên mặt phẳng với hệ toạ độ Decattes vuông góc cho n đường tròn, mỗi đường tròn xác định bởi bộ 3 số thực (X, Y, R) ở đây (X, Y) là toạ độ tâm và R là bán kính. Hai đường tròn gọi là thông nhau nếu chúng có điểm chung. Hãy chia các đường tròn thành một số tối thiểu các nhóm sao cho hai đường tròn bất kỳ trong một nhóm bất kỳ có thể đi được sang nhau sau một số hữu hạn các bước di chuyển giữa hai đường tròn thông nhau.

§5. VÀI ỨNG DỤNG CỦA CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

5.1. XÂY DỰNG CÂY KHUNG CỦA ĐỒ THỊ

Cây là đồ thị **vô hướng**, **liên thông**, **không có chu trình đơn**. Đồ thị vô hướng không có chu trình đơn gọi là rừng (hợp của nhiều cây). Như vậy mỗi thành phần liên thông của rừng là một cây.

Khái niệm cây được sử dụng rộng rãi trong nhiều lĩnh vực khác nhau: Nghiên cứu cấu trúc các phân tử hữu cơ, xây dựng các thuật toán tổ chức thư mục, các thuật toán tìm kiếm, lưu trữ và nén dữ liệu…

5.1.1. Định lý (Daisy Chain Theorem)

Giả sử T = (V, E) là đồ thị vô hướng với n đỉnh. Khi đó các mệnh đề sau là tương đương:

T là cây

T không chứa chu trình đơn và có n - 1 cạnh

T liên thông và mỗi cạnh của nó đều là cầu

Giữa hai đỉnh bất kỳ của T đều tồn tại đúng một đường đi đơn

T không chứa chu trình đơn nhưng hễ cứ thêm vào **một cạnh** ta thu được một chu trình đơn.

T liên thông và có n - 1 cạnh

Chứng minh:

1⇒2: **"T là cây" ⇒ "T không chứa chu trình đơn và có n - 1 cạnh"**

Từ T là cây, theo định nghĩa T không chứa chu trình đơn. Ta sẽ chứng minh cây T có n đỉnh thì phải có n - 1 cạnh bằng quy nạp theo số đỉnh n. Rõ ràng khi n = 1 thì cây có 1 đỉnh sẽ chứa 0 cạnh. Nếu n > 1 thì do đồ thị hữu hạn nên số các đường đi đơn trong T cũng hữu hạn, gọi P = (v_1, v_2, …, v_k) là một đường đi dài nhất (qua nhiều cạnh nhất) trong T. Đỉnh v_1 không thể có cạnh nối với đỉnh nào trong số các đỉnh v_3, v_4, …, v_k. Bởi nếu có cạnh (v_1, v_p) ($3 \le p \le k$) thì ta sẽ thiết lập được chu trình đơn (v_1, v_2, …, v_p, v_1). Mặt khác, đỉnh v_1 cũng không thể có cạnh nối với đỉnh nào khác ngoài các đỉnh trên P trên bởi nếu có cạnh (v_1, v_0) ($v_0 \notin P$) thì ta thiết lập được đường đi (v_0, v_1, v_2, …, v_k) dài hơn đường đi P. Vậy đỉnh v_1 chỉ có đúng một cạnh nối với v_2 hay v_1 là đỉnh treo. Loại bỏ v_1 và cạnh (v_1, v_2) khỏi T ta được đồ thị mới cũng là cây và có n - 1 đỉnh, cây này theo giả thiết quy nạp có n - 2 cạnh. Vậy cây T có n - 1 cạnh.

2⇒3: **"T không chứa chu trình đơn và có n - 1 cạnh"⇒"T liên thông và mỗi cạnh của nó đều là cầu"**

Giả sử T có k thành phần liên thông T_1, T_2, …, T_k. Vì T không chứa chu trình đơn nên các thành phần liên thông của T cũng không chứa chu trình đơn, tức là các T_1, T_2, …, T_k đều là cây. Gọi n_1, n_2, …, n_k lần lượt là số đỉnh của T_1, T_2, …, T_k thì cây T_1 có n_1 - 1 cạnh, cây T_2 có n_2 - 1 cạnh…, cây

T_k có n_k - 1 cạnh. Cộng lại ta có số cạnh của T là $n_1 + n_2 + \ldots + n_k - k = n - k$ cạnh. Theo giả thiết, cây T có n - 1 cạnh, suy ra k = 1, đồ thị chỉ có một thành phần liên thông là đồ thị liên thông.

Bây giờ khi T đã liên thông, nếu bỏ đi một cạnh của T thì T sẽ còn n - 2 cạnh và sẽ không liên thông bởi nếu T vẫn liên thông thì do T không có chu trình nên T sẽ là cây và có n - 1 cạnh. Điều đó chứng tỏ mỗi cạnh của T đều là cầu.

3⇒4: **"T liên thông và mỗi cạnh của nó đều là cầu"⇒"Giữa hai đỉnh bất kỳ của T có đúng một đường đi đơn"**

Gọi x và y là 2 đỉnh bất kỳ trong T, vì T liên thông nên sẽ có một đường đi đơn từ x tới y. Nếu tồn tại một đường đi đơn khác từ x tới y thì nếu ta bỏ đi một cạnh (u, v) nằm trên đường đi thứ nhất nhưng không nằm trên đường đi thứ hai thì từ u vẫn có thể đến được v bằng cách: đi từ u đi theo chiều tới x theo các cạnh thuộc đường thứ nhất, sau đó đi từ x tới y theo đường thứ hai, rồi lại đi từ y tới v theo các cạnh thuộc đường đi thứ nhất. Điều này mâu thuẫn với giả thiết (u, v) là cầu.

4⇒5: **"Giữa hai đỉnh bất kỳ của T có đúng một đường đi đơn"⇒"T không chứa chu trình đơn nhưng hễ cứ thêm vào một cạnh ta thu được một chu trình đơn"**

Thứ nhất T không chứa chu trình đơn vì nếu T chứa chu trình đơn thì chu trình đó qua ít nhất hai đỉnh u, v. Rõ ràng dọc theo các cạnh trên chu trình đó thì từ u có hai đường đi đơn tới v. Vô lý.

Giữa hai đỉnh u, v bất kỳ của T có một đường đi đơn nối u với v, vậy khi thêm cạnh (u, v) vào đường đi này thì sẽ tạo thành chu trình.

5⇒6: **"T không chứa chu trình đơn nhưng hễ cứ thêm vào một cạnh ta thu được một chu trình đơn"⇒"T liên thông và có n - 1 cạnh"**

Gọi u và v là hai đỉnh bất kỳ trong T, thêm vào T một cạnh (u, v) nữa thì theo giả thiết sẽ tạo thành một chu trình chứa cạnh (u, v). Loại bỏ cạnh này đi thì phần còn lại của chu trình sẽ là một đường đi từ u tới v. Mọi cặp đỉnh của T đều có một đường đi nối chúng tức là T liên thông, theo giả thiết T không chứa chu trình đơn nên T là cây và có n - 1 cạnh.

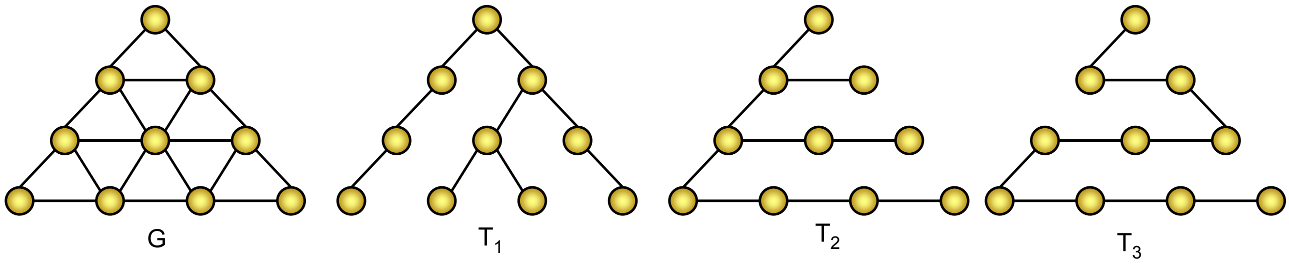
6⇒1: **"T liên thông và có n - 1 cạnh"⇒"T là cây"**

Giả sử T không là cây thì T có chu trình, huỷ bỏ một cạnh trên chu trình này thì T vẫn liên thông, nếu đồ thị mới nhận được vẫn có chu trình thì lại huỷ một cạnh trong chu trình mới. Cứ như thế cho tới khi ta nhận được một đồ thị liên thông không có chu trình. Đồ thị này là cây nhưng lại có < n - 1 cạnh (vô lý). Vậy T là cây

5.1.2. Định nghĩa

Giả sử G = (V, E) là đồ thị vô hướng. Cây T = (V, F) với $F \subset E$ gọi là cây khung của đồ thị G. Tức là nếu như loại bỏ một số cạnh của G để được một cây thì cây đó gọi là cây khung (hay cây bao trùm của đồ thị).

Dễ thấy rằng với một đồ thị vô hướng liên thông có thể có nhiều cây khung (Hình 67).



Hình 67: Đồ thị G và một số ví dụ cây khung T1, T2, T3 của nó

Điều kiện cần và đủ để một đồ thị vô hướng có cây khung là đồ thị đó phải liên thông

Số cây khung của đồ thị đầy đủ K_n là n^{n-2}.

5.1.3. Thuật toán xây dựng cây khung

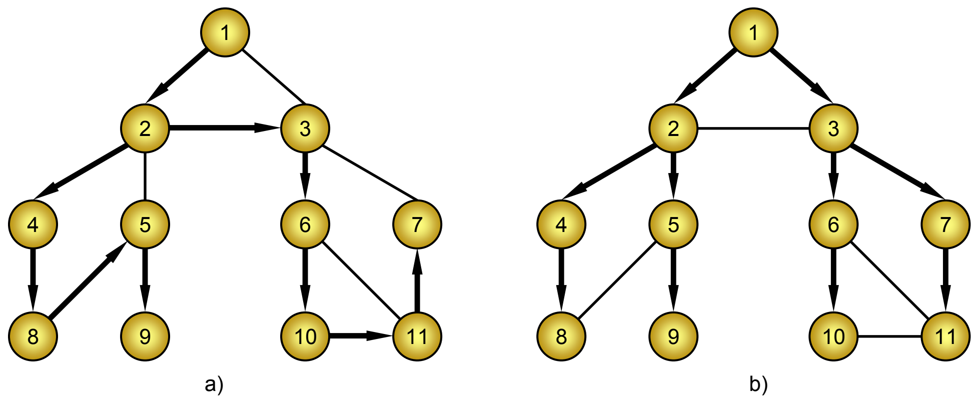
Xét đồ thị vô hướng liên thông G = (V, E) có n đỉnh, có nhiều thuật toán xây dựng cây khung của G

a) Xây dựng cây khung bằng thuật toán hợp nhất

Trước hết, đặt T = (V, ∅); T không chứa cạnh nào thì có thể coi T gồm n cây rời rạc, mỗi cây chỉ có 1 đỉnh. Sau đó xét lần lượt các cạnh của G, nếu cạnh đang xét nối hai cây khác nhau trong T thì thêm cạnh đó vào T, đồng thời hợp nhất hai cây đó lại thành một cây. Cứ làm như vậy cho tới khi kết nạp đủ n - 1 cạnh vào T thì ta được T là cây khung của đồ thị. Các phương pháp kiểm tra cạnh có nối hai cây khác nhau hay không cũng như kỹ thuật hợp nhất hai cây sẽ được bàn kỹ hơn trong thuật toán Kruskal ở §9.

b) Xây dựng cây khung bằng các thuật toán tìm kiếm trên đồ thị.

Áp dụng thuật toán BFS hay DFS bắt đầu từ đỉnh S, tại mỗi bước từ đỉnh u tới thăm đỉnh v, ta thêm vào thao tác ghi nhận luôn cạnh (u, v) vào cây khung. Do đồ thị liên thông nên thuật toán sẽ xuất phát từ S và tới thăm tất cả các đỉnh còn lại, mỗi đỉnh đúng một lần, tức là quá trình duyệt sẽ ghi nhận được đúng n - 1 cạnh. Tất cả những cạnh đó không tạo thành chu trình đơn bởi thuật toán không thăm lại những đỉnh đã thăm. Theo mệnh đề tương đương thứ hai, ta có những cạnh ghi nhận được tạo thành một cây khung của đồ thị.



Hình 68: Cây khung DFS (a) và cây khung BFS (b) (Mũi tên chỉ chiều đi thăm các đỉnh)

5.2. TẬP CÁC CHU TRÌNH CƠ BẢN CỦA ĐỒ THỊ

Xét một đồ thị vô hướng liên thông G = (V, E); gọi T = (V, F) là một cây khung của nó. Các cạnh của cây khung được gọi là các cạnh trong, còn các cạnh khác là các cạnh ngoài.

Nếu thêm một cạnh ngoài $e \in E \setminus F$ vào cây khung T, thì ta được đúng một chu trình đơn trong T, ký hiệu chu trình này là C_e. Tập các chu trình:

$$\Omega = \{C_e \mid e \in E \setminus F\}$$

được gọi là tập các chu trình cơ sở của đồ thị G.

Các tính chất quan trọng của tập các chu trình cơ sở:

Tập các chu trình cơ sở là phụ thuộc vào cây khung, **hai cây khung khác nhau có thể cho hai tập chu trình cơ sở khác nhau**.

Nếu đồ thị liên thông có n đỉnh và m cạnh, thì trong cây khung có n - 1 cạnh, còn lại m - n + 1 cạnh ngoài. Tương ứng với mỗi cạnh ngoài có một chu trình cơ sở, vậy **số chu trình cơ sở của đồ thị liên thông là m - n + 1**.

1. **Tập các chu trình cơ sở là tập nhiều nhất các chu trình** thoả mãn: Mỗi chu trình có đúng một cạnh riêng, cạnh đó không nằm trong bất cứ một chu trình nào khác. Bởi nếu có một tập gồm t chu trình thoả mãn điều đó thì việc loại bỏ cạnh riêng của một chu trình sẽ không làm mất tính liên thông của đồ thị, đồng thời không ảnh hưởng tới sự tồn tại của các chu trình khác. Như vậy nếu loại bỏ tất cả các cạnh riêng thì đồ thị vẫn liên thông và còn m - t cạnh. Đồ thị liên thông thì không thể có ít hơn n - 1 cạnh nên ta có $m - t \geq n - 1$ hay $t \leq m - n + 1$.

Mọi cạnh trong một chu trình đơn bất kỳ đều phải thuộc một chu trình cơ sở. Bởi nếu có một cạnh (u, v) không thuộc một chu trình cơ sở nào, thì khi ta bỏ cạnh đó đi đồ thị vẫn liên thông và không ảnh hưởng tới sự tồn tại của các chu trình cơ sở. Lại bỏ tiếp những cạnh ngoài của các chu trình cơ sở thì đồ thị vẫn liên thông và còn lại m - (m - n + 1) - 1 = n - 2 cạnh. Điều này vô lý.

Đối với đồ thị G = (V, E) có n đỉnh và m cạnh, có k thành phần liên thông, ta có thể xét các thành phần liên thông và xét rừng các cây khung của các thành phần đó. Khi đó có thể mở rộng khái niệm tập các chu trình cơ sở cho đồ thị vô hướng tổng quát: Mỗi khi thêm một cạnh không nằm trong các cây khung vào rừng, ta được đúng một chu trình đơn, tập các chu trình đơn tạo thành bằng cách ghép các cạnh ngoài như vậy gọi là tập các chu trình cơ sở của đồ thị G. **Số các chu trình cơ sở là m - n + k**.

5.3. ĐỊNH CHIỀU ĐỒ THỊ VÀ BÀI TOÁN LIỆT KÊ CẦU

Bài toán đặt ra là cho một đồ thị vô hướng liên thông G = (V, E), hãy thay mỗi cạnh của đồ thị bằng một cung định hướng để được một đồ thị có hướng liên thông mạnh. Nếu có phương án định chiều như vậy thì G được gọi là đồ thị định chiều được. Bài toán định chiều đồ thị có ứng dụng rõ nhất trong sơ đồ giao thông đường bộ. Chẳng hạn như trả lời câu hỏi: Trong một hệ thống đường phố,

liệu có thể quy định các đường phố đó thành đường một chiều mà vẫn đảm bảo sự đi lại giữa hai nút giao thông bất kỳ hay không.

5.3.1. Phép định chiều DFS

Xét mô hình duyệt đồ thị bằng thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh 1. Vì đồ thị là vô hướng liên thông nên quá trình tìm kiếm sẽ thăm được hết các đỉnh.

```
procedure Visit(u ∈ V);
begin
  <Thông báo thăm u và đánh dấu u đã thăm>;
  for (∀v: (u, v) ∈ E) do
    if <v chưa thăm> then Visit(v);
end;

begin
  <Đánh dấu mọi đỉnh đều chưa thăm>;
  Visit(1);
end;
```

Coi một cạnh của đồ thị tương đương với hai cung có hướng ngược chiều nhau. Thuật toán tìm kiếm theo chiều sâu theo mô hình trên sẽ duyệt qua hết các đỉnh của đồ thị và tất cả các cung nữa. Quá trình duyệt cho ta một cây tìm kiếm DFS. Ta có các nhận xét sau:

Nhận xét 1:

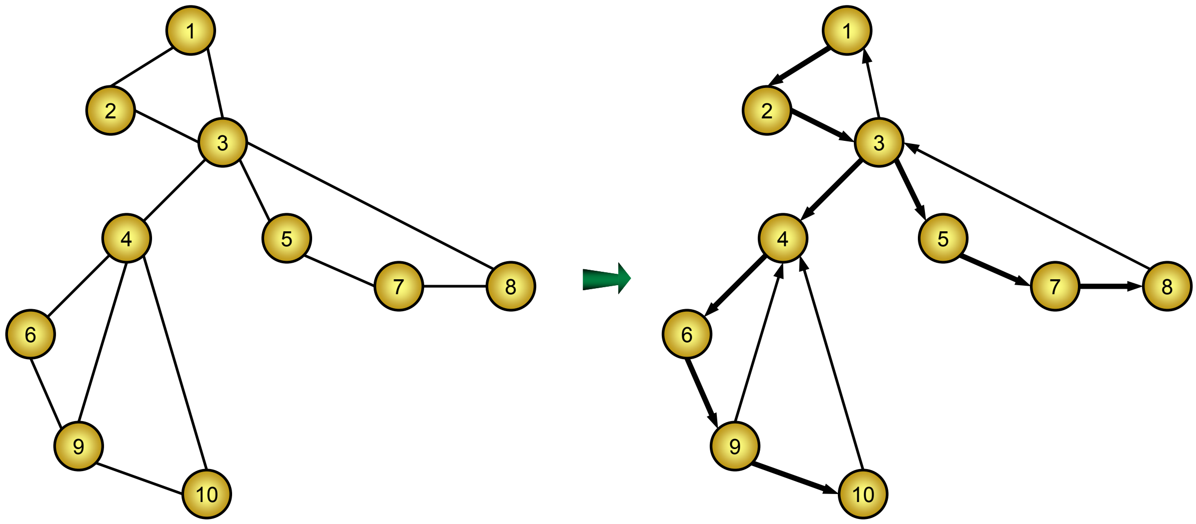
Quá trình duyệt sẽ không có cung chéo (cung đi từ một nhánh DFS thăm sau tới nhánh DFS thăm trước). Thật vậy, nếu quá trình duyệt xét tới một cung (u, v):

Nếu u thăm trước v có nghĩa là khi Visit(u) được gọi thì v chưa thăm, vì thủ tục Visit(u) sẽ xây dựng nhánh DFS gốc u gồm những đỉnh chưa thăm đến được từ u, suy ra v nằm trong nhánh DFS gốc u $\Rightarrow$ v là hậu duệ của u, hay (u, v) là cung DFS hoặc cung xuôi.

Nếu u thăm sau v (v thăm trước u), tương tự trên, ta suy ra u nằm trong nhánh DFS gốc v, v là tiền bối của u $\Rightarrow$ (u, v) là cung ngược.

Nhận xét 2:

Trong quá trình duyệt đồ thị theo chiều sâu, nếu cứ duyệt qua cung (u, v) nào thì ta bỏ đi cung (v, u). (Tức là hễ duyệt qua cung (u, v) thì ta định chiều luôn cạnh (u, v) theo chiều từ u tới v), ta được một phép định chiều đồ thị gọi là phép định chiều DFS.



Hình 69: Phép định chiều DFS

Nhận xét 3:

Với phép định chiều như trên, thì sẽ chỉ còn các cung trên cây DFS và cung ngược, không còn lại cung xuôi. Bởi trên đồ thị vô hướng ban đầu, nếu ta coi một cạnh là hai cung có hướng ngược chiều nhau thì với một cung xuôi ta có cung ngược chiều với nó là cung ngược. Do tính chất DFS, cung ngược được duyệt trước cung xuôi tương ứng, nên khi định chiều cạnh theo cung ngược thì cung xuôi sẽ bị huỷ và không bị xét tới nữa.

Nhận xét 4:

Trong đồ thị vô hướng ban đầu, cạnh bị định hướng thành cung ngược chính là cạnh ngoài của cây khung DFS. Chính vì vậy, **mọi chu trình cơ sở trong đồ thị vô hướng ban đầu vẫn sẽ là chu trình** trong đồ thị có hướng tạo ra. (Đây là một phương pháp hiệu quả để liệt kê các chu trình cơ sở của cây khung DFS: Vừa duyệt DFS vừa định chiều, nếu duyệt phải cung ngược (u, v) thì truy vết đường đi của DFS để tìm đường từ v đến u, sau đó nối thêm cung ngược (u, v) để được một chu trình cơ sở).

Định lý: Điều kiện cần và đủ để một đồ thị vô hướng liên thông có thể định chiều được là mỗi cạnh của đồ thị nằm trên ít nhất một chu trình đơn (Hay nói cách khác mọi cạnh của đồ thị đều không phải là cầu).

Chứng minh:

Gọi G = (V, E) là một đồ thị vô hướng liên thông.

"⇒"

Nếu G là định chiều được thì sau khi định hướng sẽ được đồ thị liên thông mạnh G'. Với một cạnh được định chiều thành cung (u, v) thì sẽ tồn tại một đường đi đơn trong G' theo các cạnh định hướng từ v về u. Đường đi đó nối thêm cung (u, v) sẽ thành một chu trình đơn có hướng trong G'. Tức là trên đồ thị ban đầu, cạnh (u, v) nằm trên một chu trình đơn.

"⇐"

Nếu mỗi cạnh của G đều nằm trên một chu trình đơn, ta sẽ chứng minh rằng: phép định chiều DFS sẽ tạo ra đồ thị G' liên thông mạnh.

Trước hết ta chứng minh rằng nếu (u, v) là cạnh của G thì sẽ có một đường đi từ u tới v trong G'. Thật vậy, vì (u, v) nằm trong một chu trình đơn, mà mọi cạnh của một chu trình đơn đều phải thuộc một chu trình cơ sở nào đó, nên sẽ có một chu trình cơ sở chứa cả u và v. Chu trình cơ sở qua phép định chiều DFS vẫn là chu trình trong G' nên đi theo các cạnh định hướng của chu trình đó, ta có thể đi từ u tới v và ngược lại.

Nếu u và v là 2 đỉnh bất kỳ của G thì do G liên thông, tồn tại một đường đi ($u=x_0$, x_1, …, $x_n=v$). Vì (x_i, x_{i+1}) là cạnh của G nên trong G', từ x_i có thể đến được x_{i+1}. Suy ra từ u cũng có thể đến được v bằng các cạnh định hướng của G'.

5.3.2. Cài đặt

Với những kết quả đã chứng minh trên, ta còn suy ra được: Nếu đồ thị liên thông và mỗi cạnh của nó nằm trên ít nhất một chu trình đơn thì phép định chiều DFS sẽ cho một đồ thị liên thông mạnh. Còn nếu không, thì phép định chiều DFS sẽ cho một đồ thị định hướng có ít thành phần liên thông mạnh nhất, một cạnh không nằm trên một chu trình đơn nào (cầu) của đồ thị ban đầu sẽ được định hướng thành cung nối giữa hai thành phần liên thông mạnh.

Ta sẽ cài đặt một thuật toán với một đồ thị vô hướng: liệt kê các cầu và định chiều các cạnh để được một đồ thị mới có ít thành phần liên thông mạnh nhất:

Đánh số các đỉnh theo thứ tự thăm DFS, gọi Numbering[u] là số thứ tự của đỉnh u theo cách đánh số đó. Trong quá trình tìm kiếm DFS, duyệt qua cạnh nào định chiều luôn cạnh đó. Định nghĩa thêm Low[u] là giá trị Numbering nhỏ nhất của những đỉnh đến được từ nhánh DFS gốc u bằng một cung ngược. Tức là nếu nhánh DFS gốc u có nhiều cung ngược hướng lên trên phía gốc cây thì ta ghi nhận lại cung ngược hướng lên cao nhất. Nếu nhánh DFS gốc u không chứa cung ngược thì ta cho Low[u] = +∞. Cụ thể cách cực tiểu hoá Low[u] như sau:

Trong thủ tục Visit(u), trước hết ta đánh số thứ tự thăm cho đỉnh u (Numbering[u]) và khởi gán Low[u] = +∞.

Sau đó, xét tất cả những đỉnh v kề u, định chiều cạnh (u, v) thành cung (u, v). Có hai khả năng xảy ra:

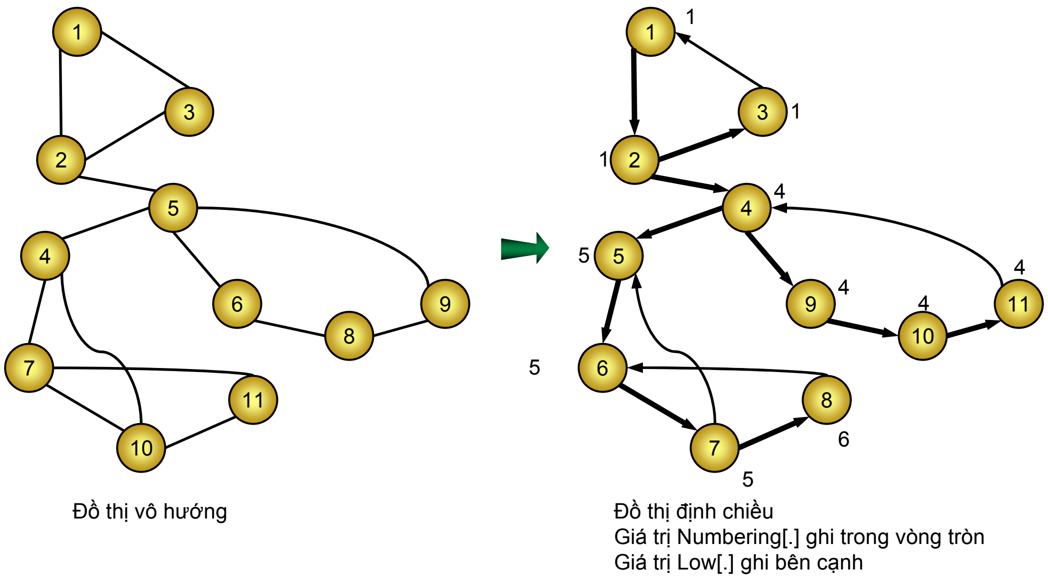
v chưa thăm thì ta gọi Visit(v) để thăm v và cực tiểu hoá Low[u] theo công thức:

$$Low[u] := min(Low[u]_{cũ}, Low[v])$$

v đã thăm thì ta cực tiểu hoá Low[u] theo công thức:

$$Low[u] := min(Low[u]_{cũ}, Numbering[v])$$

Dễ thấy cách tính như vậy là đúng đắn bởi nếu v chưa thăm thì nhánh DFS gốc v nằm trong nhánh DFS gốc u và những cung ngược trong nhánh DFS gốc v cũng là cung ngược trong nhánh DFS gốc u. Còn nếu v đã thăm thì (u, v) sẽ là cung ngược.



Hình 70: Phép đánh số và ghi nhận cung ngược lên cao nhất

Nếu từ đỉnh u tới thăm đỉnh v, (u, v) là cung DFS. Khi đỉnh v được **duyệt xong**, lùi về thủ tục Visit(u), ta so sánh Low[v] và Numbering[u]. Nếu Low[v] > Numbering[u] thì tức là nhánh DFS gốc v không có cung ngược thoát lên phía trên v. Tức là cạnh (u, v) không thuộc một chu trình cơ sở nào cả, tức cạnh đó là cầu.

```
{Đồ thị G = (V, E)}
procedure Visit(u∈V);
begin
  <Đánh số thứ tự thăm cho đỉnh u (Numbering[u]); Khởi gán Low[u] := +∞>;
  for (∀v: (u, v)∈E) do
    begin
      <Định chiều cạnh (u, v) thành cung (u, v) ⇔ Loại bỏ cung (v, u)>;
      if <v chưa thăm> then
        begin
          Visit(v);
          if Low[v] > Numbering[u] then <In ra cầu (u, v)>;
          Low[u] := Min(Low[u], Low[v]); {Cực tiểu hoá Low[u] theo Low[v]}
        end
      else {v đã thăm}
        Low[u] := Min(Low[u], Numbering[v]); {Cực tiểu hoá Low[u] theo Numbering[v]}
    end;
end;

begin
  for (∀u∈V) do
    if <u chưa thăm> then Visit(u);
  <In ra cách định chiều>;
end.
```

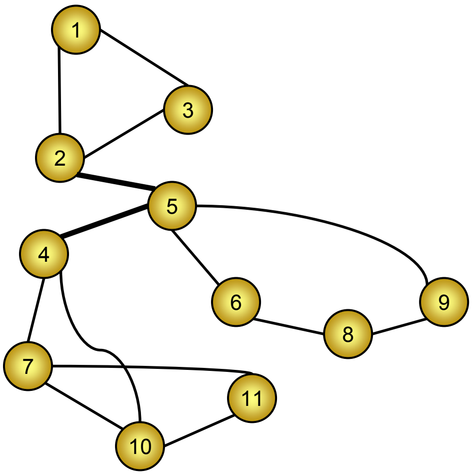
Input: file văn bản GRAPH.INP

- Dòng 1 ghi số đỉnh n (n ≤ 100) và số cạnh m của đồ thị cách nhau ít nhất một dấu cách

- m dòng tiếp theo, mỗi dòng ghi hai số nguyên dương u, v cách nhau ít nhất một dấu cách, cho biết đồ thị có cạnh nối đỉnh u với đỉnh v

Output: file văn bản BRIDGES.OUT

Thông báo các cầu và phép định chiều có ít thành phần liên thông mạnh nhất



GRAPH.INP	BRIDGES.OUT
11 14 1 2 1 3 2 3 2 5 4 5 4 7 4 10 5 6 5 9 6 8 7 10 7 11 8 9 10 11	Bridges: (5, 4) (2, 5) Directed Edges: 1 -> 2 2 -> 3 2 -> 5 3 -> 1 4 -> 7 5 -> 4 5 -> 6 6 -> 8 7 -> 10 8 -> 9 9 -> 5 10 -> 4 10 -> 11 11 -> 7

P_4_05_1.PAS * Phép định chiều DFS và liệt kê cầu

```
program Directivity_and_Bridges;
const
  InputFile  = 'GRAPH.INP';
  OutputFile = 'BRIDGES.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị}
  Numbering, Low: array[1..max] of Integer;
  n, Count: Integer;
  fo: Text;

procedure Enter;
var
  f: Text;
  i, m, u, v: Integer;
begin
  FillChar(a, SizeOf(a), False);
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n, m);
  for i := 1 to m do
    begin
      ReadLn(f, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(f);
end;

procedure Init;
begin
  FillChar(Numbering, SizeOf(Numbering), 0); {Numbering[u] = 0 ⇔ u chưa thăm}
  Count := 0;
end;
```

```
procedure Visit(u: Integer);
var
  v: Integer;
begin
  Inc(Count);
  Numbering[u] := Count; {Đánh số thứ tự thăm cho đỉnh u, u trở thành đã thăm}
  Low[u] := n + 1; {Khởi gán Low[u] bằng một giá trị đủ lớn hơn tất cả Numbering}
  for v := 1 to n do
    if a[u, v] then {Xét mọi đỉnh v kề u}
      begin
        a[v, u] := False; {Định chiều cạnh (u, v) thành cung (u, v)}
        if Numbering[v] = 0 then {Nếu v chưa thăm}
          begin
            Visit(v); {Đi thăm v}
            if Low[v] > Numbering[u] then {(u, v) là cầu}
              WriteLn(fo, '(', u, ', ', v, ')');
            if Low[u] > Low[v] then Low[u] := Low[v]; {Cực tiểu hoá Low[u] }
          end
        else
          if Low[u] > Numbering[v] then Low[u] := Numbering[v]; {Cực tiểu hoá Low[u] }
      end;
end;

procedure Solve;
var
  u, v: Integer;
begin
  WriteLn(fo, 'Bridges: '); {Dùng DFS để định chiều đồ thị và liệt kê cầu}
  for u := 1 to n do
    if Numbering[u] = 0 then Visit(u);
  WriteLn(fo, 'Directed Edges: '); {Quét lại ma trận kề để in ra các cạnh định hướng}
  for u := 1 to n do
    for v := 1 to n do
      if a[u, v] then WriteLn(fo, u, ' -> ', v);
end;

begin
  Enter;
  Assign(fo, OutputFile); Rewrite(fo);
  Init;
  Solve;
  Close(fo);
end.
```

5.4. LIỆT KÊ KHỚP

Trong đồ thị vô hướng, Một đỉnh C được gọi là khớp, nếu như ta bỏ đi đỉnh C và các cạnh liên thuộc với nó thì sẽ làm tăng số thành phần liên thông của đồ thị. Bài toán đặt ra là phải liệt kê hết các khớp của đồ thị.

Rõ ràng theo cách định nghĩa trên, các đỉnh treo và đỉnh cô lập sẽ không phải là khớp. Đồ thị liên thông có ≥ 3 đỉnh, không có khớp (cho dù bỏ đi đỉnh nào đồ thị vẫn liên thông) được gọi là đồ thị song liên thông. Giữa hai đỉnh phân biệt của đồ thị song liên thông, tồn tại ít nhất 2 đường đi không có đỉnh trung gian nào chung.

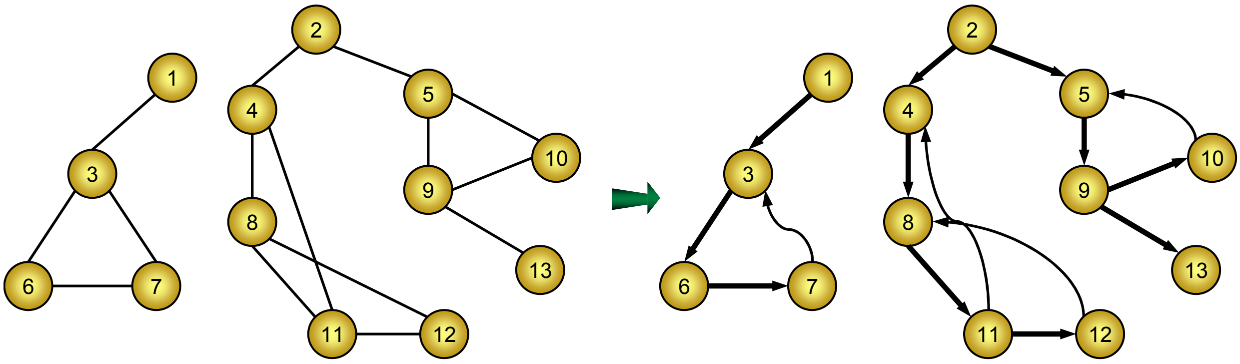
Coi mỗi cạnh của đồ thị ban đầu là hai cung có hướng ngược chiều nhau và dùng phép duyệt đồ thị theo chiều sâu:

```
{Đồ thị G = (V, E)}
```

```
procedure Visit(u ∈ V): ∈ V;
begin
  <Thông báo thăm u và đánh dấu u đã thăm>;
  for (∀v: (u, v) ∈ E) do
    if <v chưa thăm> then Visit(v);
end;

begin
  <Đánh dấu mọi đỉnh đều chưa thăm>;
  for (∀u∈V) do
    if <u chưa thăm> then Visit(u);
end;
```

Quá trình duyệt cho một rừng các cây DFS. Các cung duyệt qua có ba loại: cung DFS, cung ngược và cung xuôi, để không bị rối hình, ta chỉ ưu tiên vẽ cung DFS hoặc cung ngược:



Hình 71 Duyệt DFS, xác định cây DFS và các cung ngược

Hãy để ý nhánh DFS gốc ở đỉnh r nào đó

Nếu **mọi** nhánh con của nhánh DFS gốc r đều có một cung ngược lên tới một tiền bối của r thì r không là khớp. Bởi nếu trong đồ thị ban đầu, ta bỏ r đi thì từ mỗi đỉnh bất kỳ của nhánh con, ta vẫn có thể đi lên một tiền bối của r, rồi đi sang nhánh con khác hoặc đi sang tất cả những đỉnh còn lại của cây. Số thành phần liên thông của đồ thị không thay đổi.

Nếu r không phải là gốc của một cây DFS, và **tồn tại** một nhánh con của nhánh DFS gốc r không có cung ngược lên một tiền bối của r thì r là khớp. Bởi khi đó, tất cả những cung xuất phát từ nhánh con đó chỉ đi tới những đỉnh nội bộ trong nhánh DFS gốc r mà thôi, trên đồ thị ban đầu, không tồn tại cạnh nối từ những đỉnh thuộc nhánh con tới một tiền bối của r. Vậy từ nhánh đó muốn đi lên một tiền bối của r, tất phải đi qua r. Huỷ r khỏi đồ thị sẽ làm mất tất cả các đường đi đó, tức là làm tăng số thành phần liên thông của đồ thị.

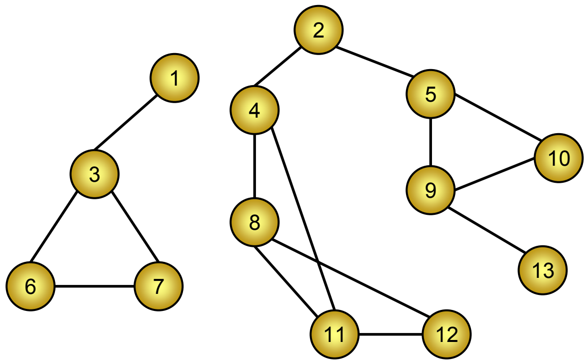
Nếu r là gốc của một cây DFS, thì r là khớp khi và chỉ khi r có ít nhất hai nhánh con. Bởi khi r có 2 nhánh con thì đường đi giữa hai đỉnh thuộc hai nhánh con đó tất phải đi qua r.

Vậy thì thuật toán liệt kê khớp lại là những kỹ thuật quen thuộc, duyệt DFS, đánh số, ghi nhận cạnh ngược lên cao nhất từ một nhánh con, chỉ thêm vào đó một thao tác nhỏ: Nếu từ đỉnh u gọi đệ quy thăm đỉnh v ((u, v) là cung DFS) thì sau khi duyệt xong đỉnh v, lùi về thủ tục Visit(u), ta so sánh Low[v] và Numbering[u] để kiểm tra xem từ nhánh con gốc v có cạnh ngược nào lên tiền bối của u hay không, nếu không có thì tạm thời đánh dấu u là khớp. Cuối cùng phải kiểm

tra lại điều kiện: nếu u là gốc cây DFS thì nó là khớp khi và chỉ khi nó có ít nhất 2 nhánh con, nếu không thoả mãn điều kiện đó thì đánh dấu lại u không là khớp.

Input: file văn bản GRAPH.INP với khuôn dạng như bài toán liệt kê cầu

Output: file văn bản CUTV.OUT ghi các khớp của đồ thị



GRAPH.INP	CUTV.OUT
13 15 1 3 2 4 2 5 3 6 3 7 4 8 4 11 5 9 5 10 6 7 8 11 8 12 9 10 9 13 11 12	Cut vertices: 2, 3, 4, 5, 9,

P_4_05_2.PAS * Liệt kê các khớp của đồ thị

```
program CutVertices;
const
  InputFile  = 'GRAPH.INP';
  OutputFile = 'CUTV.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị}
  Numbering, Low, nC: array[1..max] of Integer; {nC[u]: Số nhánh con của nhánh DFS gốc u}
  Mark: array[1..max] of Boolean; {Mark[u] = True ⇔ u là khớp}
  n, Count: Integer;

procedure LoadGraph;
var
  i, m, u, v: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  FillChar(a, SizeOf(a), False);
  ReadLn(f, n, m);
  for i := 1 to m do
    begin
      ReadLn(f, u, v);
      a[u, v] := True; a[v, u] := True;
    end;
  Close(f);
end;

procedure Visit(u: Integer); {Tìm kiếm theo chiều sâu bắt đầu từ u}
var
  v: Integer;
begin
  Inc(Count);
  Numbering[u] := Count; Low[u] := n + 1; nC[u] := 0;
  Mark[u] := False;
  for v := 1 to n do
    if a[u, v] then {Xét mọi v kề u}
```

```
      if Numbering[v] = 0 then {Nếu v chưa thăm}
        begin
          Inc(nc[u]); {Tăng biến đếm số con của u lên 1}
          Visit(v); {Thăm v}
       {Nếu nhánh DFS gốc v không có cung ngược lên một tiền bối của u tức là Low[v] ≥ Numbering[u]}
          Mark[u] := Mark[u] or (Low[v] >= Numbering[u]); {Tạm đánh dấu u là khớp}
          if Low[u] > Low[v] then Low[u] := Low[v]; {Cực tiểu hoá Low[u] }
        end
      else
        if Low[u] > Numbering[v] then Low[u] := Numbering[v]; {Cực tiểu hoá Low[u] }
end;

procedure Solve;
var
  u: Integer;
begin
  FillChar(Numbering, SizeOf(Numbering), 0); {Đánh số = 0 ⇔ Đỉnh chưa thăm}
  FillChar(Mark, SizeOf(Mark), False); {Mảng đánh dấu khớp chưa có gì}
  Count := 0;
  for u := 1 to n do
    if Numbering[u] = 0 then {Xét mọi đỉnh u chưa thăm}
      begin
        Visit(u); {Thăm u, xây dựng cây DFS gốc u}
        if nC[u] < 2 then {Nếu u có ít hơn 2 con}
          Mark[u] := False; {Thì u không phải là khớp}
      end;
end;

procedure Result; {Dựa vào mảng đánh dấu để liệt kê các khớp}
var
  i: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  WriteLn(f, 'Cut vertices:');
  for i := 1 to n do
    if Mark[i] then Write(f, i, ', ');
  Close(f);
end;

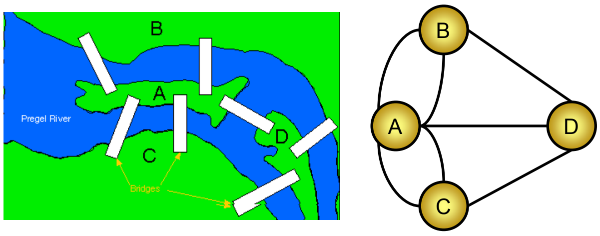
begin
  LoadGraph;
  Solve;
  Result;
end.
```

§6. CHU TRÌNH EULER, ĐƯỜNG ĐI EULER, ĐỒ THỊ EULER

6.1. BÀI TOÁN 7 CÁI CẦU

Thành phố Konigsberg thuộc Phổ (nay là Kaliningrad thuộc Cộng hoà Nga), được chia làm 4 vùng bằng các nhánh sông Pregel. Các vùng này gồm 2 vùng bên bờ sông (B, C), đảo Kneiphof (A) và một miền nằm giữa hai nhánh sông Pregel (D). Vào thế kỷ XVIII, người ta đã xây 7 chiếc cầu nối những vùng này với nhau. Người dân ở đây tự hỏi: Liệu có cách nào xuất phát tại một địa điểm trong thành phố, đi qua 7 chiếc cầu, mỗi chiếc đúng 1 lần rồi quay trở về nơi xuất phát không ?

Nhà toán học Thụy sĩ Leonhard Euler đã giải bài toán này và có thể coi đây là ứng dụng đầu tiên của Lý thuyết đồ thị, ông đã mô hình hoá sơ đồ 7 cái cầu bằng một đa đồ thị, bốn vùng được biểu diễn bằng 4 đỉnh, các cầu là các cạnh. Bài toán tìm đường qua 7 cầu, mỗi cầu đúng một lần có thể tổng quát hoá bằng bài toán: **Có tồn tại chu trình đơn trong đa đồ thị chứa tất cả các cạnh ?.**



Hình 72: Mô hình đồ thị của bài toán bảy cái cầu

6.2. ĐỊNH NGHĨA

Chu trình đơn chứa tất cả các cạnh của đồ thị được gọi là chu trình Euler

Đường đi đơn chứa tất cả các cạnh của đồ thị được gọi là đường đi Euler

Một đồ thị có chu trình Euler được gọi là đồ thị Euler

Một đồ thị có đường đi Euler được gọi là đồ thị nửa Euler.

Rõ ràng một đồ thị Euler thì phải là nửa Euler nhưng điều ngược lại thì không phải luôn đúng

6.3. ĐỊNH LÝ

Một đồ thị **vô hướng** **<u>liên thông</u>** G = (V, E) có **chu trình Euler** khi và chỉ khi mọi đỉnh của nó đều có bậc chẵn: $deg(v) \equiv 0 \pmod 2$ $(\forall v \in V)$

Một đồ thị vô hướng liên thông **có đường đi Euler nhưng không có chu trình Euler** khi và chỉ khi nó có đúng 2 đỉnh bậc lẻ

Một đồ thi **có hướng** **<u>liên thông yếu</u>** G = (V, E) có **chu trình Euler** thì mọi đỉnh của nó có bán bậc ra bằng bán bậc vào: $deg^+(v) = deg^-(v)$ $(\forall v \in V)$; Ngược lại, nếu G **liên thông yếu** và mọi đỉnh của nó có bán bậc ra bằng bán bậc vào thì G có **chu trình Euler**, hay G sẽ là **liên thông mạnh.**

Một đồ thị có hướng liên thông yếu G = (V, E) có **đường đi Euler nhưng không có chu trình Euler** nếu tồn tại đúng hai đỉnh u, v ∈ V sao cho deg+(u) - deg-(u) = deg-(v) - deg+(v) = 1, còn tất cả những đỉnh khác u và v đều có bán bậc ra bằng bán bậc vào.

6.4. THUẬT TOÁN FLEURY TÌM CHU TRÌNH EULER

6.4.1. Đối với đồ thị vô hướng liên thông, mọi đỉnh đều có bậc chẵn.

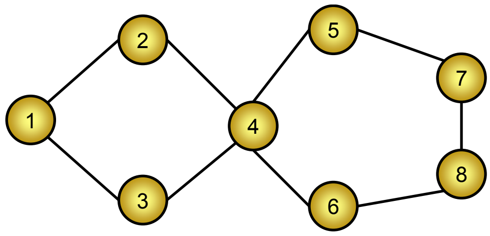
Xuất phát từ một đỉnh, ta chọn một cạnh liên thuộc với nó để đi tiếp theo hai nguyên tắc sau:

Xoá bỏ cạnh đã đi qua

Chỉ đi qua cầu khi không còn cạnh nào khác để chọn

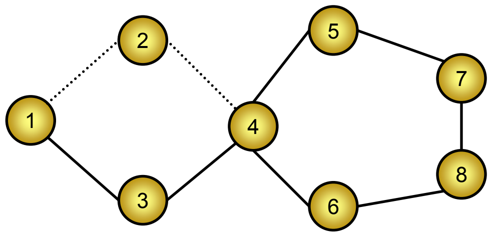
Và ta cứ chọn cạnh đi một cách thoải mái như vậy cho tới khi không đi tiếp được nữa, đường đi tìm được là chu trình Euler.

Ví dụ: Với đồ thị ở Hình 73:



Hình 73

Nếu xuất phát từ đỉnh 1, có hai cách đi tiếp: hoặc sang 2 hoặc sang 3, giả sử ta sẽ sang 2 và xoá cạnh (1, 2) vừa đi qua. Từ 2 chỉ có cách duy nhất là sang 4, nên cho dù (2, 4) là cầu ta cũng phải đi sau đó xoá luôn cạnh (2, 4). Đến đây, các cạnh còn lại của đồ thị có thể vẽ như Hình 74 bằng nét liền, các cạnh đã bị xoá được vẽ bằng nét đứt.



Hình 74

Bây giờ đang đứng ở đỉnh 4 thì ta có 3 cách đi tiếp: sang 3, sang 5 hoặc sang 6. Vì (4, 3) là cầu nên ta sẽ không đi theo cạnh (4, 3) mà sẽ đi (4, 5) hoặc (4, 6). Nếu đi theo (4, 5) và cứ tiếp tục đi như vậy, ta sẽ được chu trình Euler là (1, 2, 4, 5, 7, 8, 6, 4, 3, 1). Còn đi theo (4, 6) sẽ tìm được chu trình Euler là: (1, 2, 4, 6, 8, 7, 5, 4, 3, 1).

6.4.2. Đối với đồ thị có hướng liên thông yếu, mọi đỉnh đều có bán bậc ra bằng bán bậc vào.

Bằng cách "lạm dụng thuật ngữ", ta có thể mô tả được thuật toán tìm chu trình Euler cho cả đồ thị có hướng cũng như vô hướng:

Thứ nhất, dưới đây nếu ta nói cạnh (u, v) thì hiểu là cạnh nối đỉnh u và đỉnh v trên đồ thị vô hướng, hiểu là cung nối từ đỉnh u tới đỉnh v trên đồ thị có hướng.

Thứ hai, ta gọi cạnh (u, v) là "một đi không trở lại" nếu như từ u ta đi tới v theo cạnh đó, sau đó xoá cạnh đó đi thì không có cách nào từ v quay lại u.

Vậy thì thuật toán Fleury tìm chu trình Euler có thể mô tả như sau:

Xuất phát từ một đỉnh, ta đi một cách tuỳ ý theo các cạnh tuân theo hai nguyên tắc: Xoá bỏ cạnh vừa đi qua và chỉ chọn cạnh "một đi không trở lại" nếu như không còn cạnh nào khác để chọn.

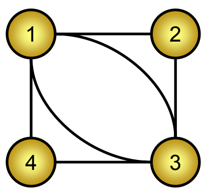
6.5. CÀI ĐẶT

Ta sẽ cài đặt thuật toán Fleury trên một đa đồ thị vô hướng. Để đơn giản, ta coi đồ thị này đã có chu trình Euler, công việc của ta là tìm ra chu trình đó thôi. Bởi việc kiểm tra tính liên thông cũng như kiểm tra mọi đỉnh đều có bậc chẵn đến giờ có thể coi là chuyện nhỏ.

Input: file văn bản EULER.INP

- Dòng 1: Chứa số đỉnh n của đồ thị ($n \le 100$)
- Các dòng tiếp theo, mỗi dòng chứa 3 số nguyên dương cách nhau ít nhất 1 dấu cách có dạng: u v k cho biết giữa đỉnh u và đỉnh v có k cạnh nối

Output: file văn bản EULER.OUT, ghi chu trình EULER



EULER.INP	EULER.OUT
5 1 2 1 1 3 2 1 4 1 2 3 1 3 4 1	1 2 3 1 3 4 1

P_4_06_1.PAS * Thuật toán Fleury tìm chu trình Euler

```
program Euler_Circuit;
const
  InputFile  = 'EULER.INP';
  OutputFile = 'EULER.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Integer;
  n: Integer;

procedure Enter;
var
  u, v, k: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  FillChar(a, SizeOf(a), 0);
  ReadLn(f, n);
```

```
  while not SeekEof(f) do
    begin
      ReadLn(f, u, v, k);
      a[u, v] := k;
      a[v, u] := k;
    end;
  Close(f);
end;

{Thủ tục này kiểm tra nếu xoá một cạnh nối (x, y) thì y có còn quay lại được x hay không}
function CanGoBack(x, y: Integer): Boolean;
var
  Queue: array[1..max] of Integer; {Hàng đợi dùng cho Breadth First Search}
  First, Last: Integer; {First: Chỉ số đầu hàng đợi, Last: Chỉ số cuối hàng đợi}
  u, v: Integer;
  Free: array[1..max] of Boolean; {Mảng đánh dấu}
begin
  Dec(a[x, y]); Dec(a[y, x]); {Thử xoá một cạnh (x, y) ⇔ Số cạnh nối (x, y) giảm 1}
  FillChar(Free, n, True); {sau đó áp dụng BFS để xem từ y có quay lại x được không ?}
  Free[y] := False;
  First := 1; Last := 1;
  Queue[1] := y;
  repeat
    u := Queue[First]; Inc(First);
    for v := 1 to n do
      if Free[v] and (a[u, v] > 0) then
        begin
          Inc(Last);
          Queue[Last] := v;
          Free[v] := False;
          if Free[x] then Break;
        end;
  until First > Last;
  CanGoBack := not Free[x];
  Inc(a[x, y]); Inc(a[y, x]); {ở trên đã thử xoá cạnh thì giờ phải phục hồi}
end;

procedure FindEulerCircuit; {Thuật toán Fleury}
var
  Current, Next, v, count: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  Current := 1;
  Write(f, 1, ' '); {Bắt đầu từ đỉnh Current = 1}
  count := 1;
  repeat
    Next := 0;
    for v := 1 to n do
      if a[Current, v] > 0 then
        begin
          Next := v;
          if CanGoBack(Current, Next) then Break;
        end;
    if Next <> 0 then
      begin
        Dec(a[Current, Next]);
        Dec(a[Next, Current]); {Xoá bỏ cạnh vừa đi qua}
        Write(f, Next, ' '); {In kết quả đi tới Next}
        Inc(count);
        if count mod 16 = 0 then WriteLn; {In ra tối đa 16 đỉnh trên một dòng}
        Current := Next; {Lại tiếp tục với đỉnh đang đứng là Next}
      end;
  until Next = 0; {Cho tới khi không đi tiếp được nữa}
```

```
  Close(f);
end;

begin
  Enter;
  FindEulerCircuit;
end.
```

6.6. THUẬT TOÁN TỐT HƠN

Trong trường hợp đồ thị Euler có **số cạnh đủ nhỏ**, ta có thể sử dụng phương pháp sau để tìm chu trình Euler trong đồ thị vô hướng: Bắt đầu từ một chu trình đơn C bất kỳ, chu trình này tìm được bằng cách xuất phát từ một đỉnh, đi tuỳ ý theo các cạnh cho tới khi quay về đỉnh xuất phát, lưu ý là đi qua cạnh nào xoá luôn cạnh đó. Nếu như chu trình C tìm được chứa tất cả các cạnh của đồ thị thì đó là chu trình Euler. Nếu không, xét các đỉnh dọc theo chu trình C, nếu còn có cạnh chưa xoá liên thuộc với một đỉnh u nào đó thì lại từ u, ta đi tuỳ ý theo các cạnh cũng theo nguyên tắc trên cho tới khi quay trở về u, để được một chu trình đơn khác qua u. Loại bỏ vị trí u khỏi chu trình C và chèn vào C chu trình mới tìm được tại đúng vị trí của u vừa xoá, ta được một chu trình đơn C' mới lớn hơn chu trình C. Cứ làm như vậy cho tới khi được chu trình Euler. Việc chứng minh tính đúng đắn của thuật toán cũng là chứng minh định lý về điều kiện cần và đủ để một đồ thị vô hướng liên thông có chu trình Euler.

Mô hình thuật toán có thể viết như sau:

```
<Khởi tạo một ngăn xếp Stack ban đầu chỉ gồm mỗi đỉnh 1>;
<Mô tả các phương thức Push (đẩy vào) và Pop(lấy ra) một đỉnh từ ngăn xếp Stack, phương thức Get cho biết phần tử nằm ở đỉnh Stack. Khác với Pop, phương thức Get chỉ cho biết phần tử ở đỉnh Stack chứ không lấy phần tử đó ra>;
while Stack ≠∅ do
begin
  x := Get;
  if <Tồn tại đỉnh y mà (x, y)∈E> then {Từ x còn đi hướng khác được}
    begin
      Push(y);
      <Loại bỏ cạnh (x, y) khỏi đồ thị>;
    end
  else {Từ x không đi tiếp được tới đâu nữa}
    begin
      x := Pop;
      <In ra đỉnh x trên đường đi Euler>;
    end;
end;
```

Thuật toán trên có thể dùng để tìm chu trình Euler trong đồ thị có hướng liên thông yếu, mọi đỉnh có bán bậc ra bằng bán bậc vào. Tuy nhiên thứ tự các đỉnh in ra bị ngược so với các cung định hướng, ta có thể đảo ngược hướng các cung trước khi thực hiện thuật toán để được thứ tự đúng.

Thuật toán hoạt động với hiệu quả cao, dễ cài đặt, nhưng trường hợp xấu nhất thì Stack sẽ phải chứa toàn bộ danh sách đỉnh trên chu trình Euler chính vì vậy mà khi đa đồ thị có số cạnh quá lớn thì sẽ không đủ không gian nhớ mô tả Stack (Ta cứ thử với đồ thị chỉ gồm 2 đỉnh nhưng giữa hai đỉnh đó có tới 10^6 cạnh nối sẽ thấy ngay). Lý do thuật toán chỉ có thể áp dụng trong trường hợp số cạnh có giới hạn biết trước đủ nhỏ là như vậy.

P_4_06_2.PAS * Thuật toán hiệu quả tìm chu trình Euler

```
program Euler_Circuit;
const
  InputFile  = 'EULER.INP';
  OutputFile = 'EULER.OUT';
  max = 100;
  maxE = 20000; {Số cạnh tối đa}
var
  a: array[1..max, 1..max] of Integer;
  stack: array[1..maxE] of Integer;
  n, last: Integer;

procedure Enter; {Nhập dữ liệu}
var
  u, v, k: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  FillChar(a, SizeOf(a), 0);
  ReadLn(f, n);
  while not SeekEof(f) do
    begin
      ReadLn(f, u, v, k);
      a[u, v] := k;
      a[v, u] := k;
    end;
  Close(f);
end;

procedure Push(v: Integer); {Đẩy một đỉnh v vào ngăn xếp}
begin
  Inc(last);
  Stack[last] := v;
end;

function Pop: Integer; {Lấy một đỉnh khỏi ngăn xếp, trả về trong kết quả hàm}
begin
  Pop := Stack[last];
  Dec(last);
end;

function Get: Integer; {Trả về phần tử ở đỉnh (Top) ngăn xếp}
begin
  Get := Stack[last];
end;

procedure FindEulerCircuit;
var
  u, v, count: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  Stack[1] := 1; {Khởi tạo ngăn xếp ban đầu chỉ gồm đỉnh 1}
  last := 1;
  count := 0;
  while last <> 0 do {Chừng nào ngăn xếp chưa rỗng}
    begin
      u := Get; {Xác định u là phần tử ở đỉnh ngăn xếp}
      for v := 1 to n do
        if a[u, v] > 0 then {Xét tất cả các cạnh liên thuộc với u, nếu thấy}
          begin
            Dec(a[u, v]); Dec(a[v, u]); {Xoá cạnh đó khỏi đồ thị}
            Push(v); {Đẩy đỉnh tiếp theo vào ngăn xếp}
```

```
            Break;
          end;
      if u = Get then {Nếu phần tử ở đỉnh ngăn xếp vẫn là u ⇒ vòng lặp trên không tìm thấy đỉnh nào kề với u}
        begin
          Inc(count);
          Write(f, Pop, ' '); {In ra phần tử đỉnh ngăn xếp}
        end;
    end;
  Close(f);
end;

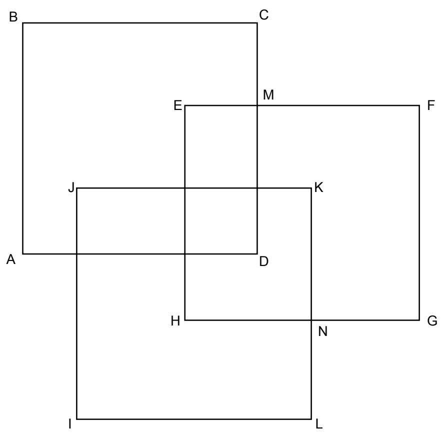
begin
  Enter;
  FindEulerCircuit;
end.
```

Bài tập

Trên mặt phẳng cho n hình chữ nhật có các cạnh song song với các trục toạ độ. Hãy chỉ ra một chu trình:

Chỉ đi trên cạnh của các hình chữ nhật

Trên cạnh của mỗi hình chữ nhật, ngoại trừ những giao điểm với cạnh của hình chữ nhật khác có thể qua nhiều lần, những điểm còn lại chỉ được qua đúng một lần.



M D A B C M F G N L I J K N H E M

§7. CHU TRÌNH HAMILTON, ĐƯỜNG ĐI HAMILTON, ĐỒ THỊ HAMILTON

7.1. ĐỊNH NGHĨA

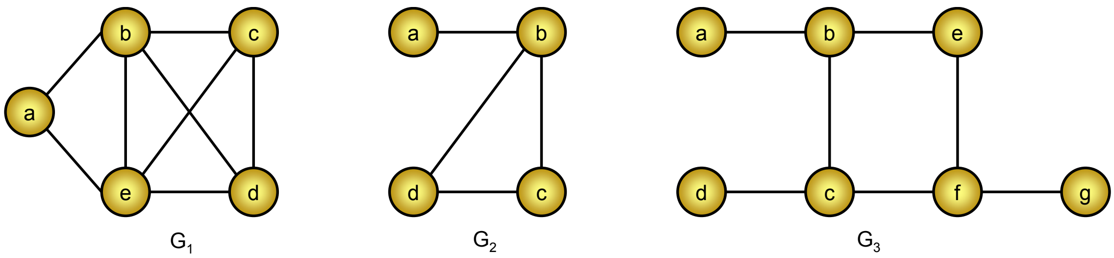
Cho đồ thị G = (V, E) có n đỉnh

Chu trình (x_1, x_2, …, x_n, x_1) được gọi là chu trình Hamilton nếu $x_i \neq x_j$ với $1 \leq i < j \leq n$

Đường đi (x_1, x_2, …, x_n) được gọi là đường đi Hamilton nếu $x_i \neq x_j$ với $1 \leq i < j \leq n$

Có thể phát biểu một cách hình thức: Chu trình Hamilton là chu trình xuất phát từ 1 đỉnh, đi thăm tất cả những đỉnh còn lại mỗi đỉnh đúng 1 lần, cuối cùng quay trở lại đỉnh xuất phát. Đường đi Hamilton là đường đi qua tất cả các đỉnh của đồ thị, mỗi đỉnh đúng 1 lần. Khác với khái niệm chu trình Euler và đường đi Euler, một chu trình Hamilton không phải là đường đi Hamilton bởi có đỉnh xuất phát được thăm tới 2 lần.

Ví dụ: Xét 3 đơn đồ thị G_1, G_2, G_3 như trong Hình 75:



Hình 75

Đồ thị G_1 có chu trình Hamilton (a, b, c, d, e, a). G_2 không có chu trình Hamilton vì deg(a) = 1 nhưng có đường đi Hamilton (a, b, c, d). G_3 không có cả chu trình Hamilton lẫn đường đi Hamilton

7.2. ĐỊNH LÝ

Đồ thị vô hướng G, trong đó tồn tại k đỉnh sao cho nếu xoá đi k đỉnh này cùng với những cạnh liên thuộc của chúng thì đồ thị nhận được sẽ có nhiều hơn k thành phần liên thông. Thì khẳng định là G không có chu trình Hamilton. Mệnh đề phản đảo của định lý này cho ta điều kiện cần để một đồ thị có chu trình Hamilton

Định lý Dirac (1952): Đồ thị vô hướng G có n đỉnh ($n \geq 3$). Khi đó nếu mọi đỉnh v của G đều có $deg(v) \geq n/2$ thì G có chu trình Hamilton. Đây là một điều kiện đủ để một đồ thị có chu trình Hamilton.

Đồ thị có hướng G liên thông mạnh và có n đỉnh. Nếu $deg^+(v) \geq n / 2$ và $deg^-(v) \geq n / 2$ với mọi đỉnh v thì G có chu trình Hamilton

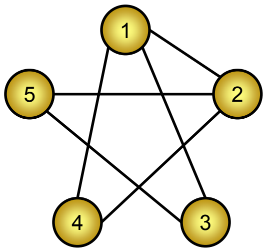
7.3. CÀI ĐẶT

Dưới đây ta sẽ cài đặt một chương trình liệt kê tất cả các chu trình Hamilton xuất phát từ đỉnh 1, các chu trình Hamilton khác có thể có được bằng cách hoán vị vòng quanh. Lưu ý rằng cho tới nay, người ta vẫn **chưa tìm ra** một phương pháp nào thực sự hiệu quả hơn phương pháp quay lui để tìm dù chỉ một chu trình Hamilton cũng như đường đi Hamilton trong trường hợp đồ thị tổng quát.

Input: file văn bản HAMILTON.INP

- Dòng 1 ghi số đỉnh n ($2 \le n \le 100$) và số cạnh m của đồ thị cách nhau 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng hai số nguyên dương u, v cách nhau 1 dấu cách, thể hiện u, v là hai đỉnh kề nhau trong đồ thị

Output: file văn bản HAMILTON.OUT liệt kê các chu trình Hamilton



HAMILTON.INP	HAMILTON.OUT
5 6 1 2 1 3 2 4 3 5 4 1 5 2	1 3 5 2 4 1 1 4 2 5 3 1

P_4_07_1.PAS * Thuật toán quay lui liệt kê chu trình Hamilton

```
program All_of_Hamilton_Circuits;
const
  InputFile  = 'HAMILTON.INP';
  OutputFile = 'HAMILTON.OUT';
  max = 100;
var
  fo: Text;
  a: array[1..max, 1..max] of Boolean; {Ma trận kề của đồ thị: a[u, v] = True ⇔ (u, v) là cạnh}
  Free: array[1..max] of Boolean; {Mảng đánh dấu Free[v] = True nếu chưa đi qua đỉnh v}
  X: array[1..max] of Integer; {Chu trình Hamilton sẽ tìm là; 1=X[1]→X[2] → … →X[n] →X[1]=1}
  n: Integer;

procedure Enter;
var
  i, u, v, m: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  FillChar(a, SizeOf(a), False);
  ReadLn(f, n, m);
  for i := 1 to m do
    begin
      ReadLn(f, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(f);
end;

procedure PrintResult; {In kết quả nếu tìm thấy chu trình Hamilton}
var
  i: Integer;
begin
  for i := 1 to n do Write(fo, X[i], ' ');
  WriteLn(fo, X[1]);
```

```
end;

procedure Try(i: Integer); {Thử các cách chọn đỉnh thứ i trong hành trình}
var
  j: Integer;
begin
  for j := 1 to n do     {Đỉnh thứ i (X[i]) có thể chọn trong những đỉnh}
    if Free[j] and a[x[i - 1], j] then {kề với X[i - 1] và chưa bị đi qua }
      begin
        x[i] := j; {Thử một cách chọn X[i]}
        if i < n then     {Nếu chưa thử chọn đến X[n]}
          begin
            Free[j] := False; {Đánh dấu đỉnh j là đã đi qua}
            Try(i + 1); {Để các bước thử kế tiếp không chọn phải đỉnh j nữa}
            Free[j] := True; {Sẽ thử phưng án khác cho X[i] nên sẽ bỏ đánh dấu đỉnh vừa thử}
          end
        else {Nếu đã thử chọn đến X[n]}
          if a[j, X[1]] then PrintResult; {và nếu X[n] lại kề với X[1] thì ta có chu trình Hamilton}
      end;
end;

begin
  Enter;
  FillChar(Free, SizeOf(Free), True); {Khởi tạo: Các đỉnh đều chưa đi qua}
  x[1] := 1; Free[1] := False; {Bắt đầu từ đỉnh 1}
  Assign(fo, OutputFile); Rewrite(fo);
  Try(2); {Thử các cách chọn đỉnh kế tiếp}
  Close(fo);
end.
```

Bài tập

Bài 1

a) Lập chương trình nhập vào một đồ thị và chỉ ra đúng một chu trình Hamilton nếu có.

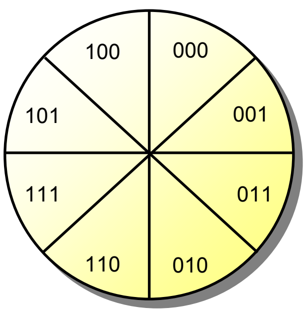
b) Lập chương trình nhập vào một đồ thị và chỉ ra đúng một đường đi Hamilton nếu có.

Bài 2

Trong đám cưới của Péc-xây và An-đrơ-nét có 2n hiệp sỹ. Mỗi hiệp sỹ có không quá n - 1 kẻ thù. Hãy giúp Ca-xi-ô-bê, mẹ của An-đrơ-nét xếp 2n hiệp sỹ ngồi quanh một bàn tròn sao cho không có hiệp sỹ nào phải ngồi cạnh kẻ thù của mình. Mỗi hiệp sỹ sẽ cho biết những kẻ thù của mình khi họ đến sân rồng.

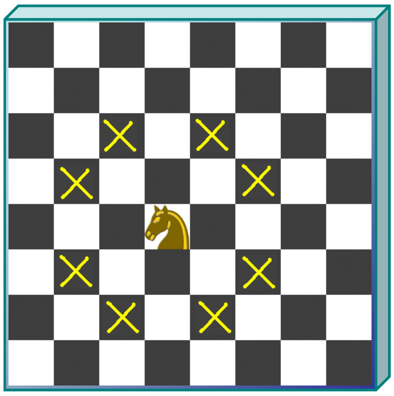
Bài 3

Gray code: Một hình tròn được chia thành 2^n hình quạt đồng tâm. Hãy xếp tất cả các xâu nhị phân độ dài n vào các hình quạt, mỗi xâu vào một hình quạt sao cho bất cứ hai xâu nào ở hai hình quạt cạnh nhau đều chỉ khác nhau đúng 1 bit. Ví dụ với n = 3:



Bài 4

Thách đố: Bài toán mã đi tuần: Trên bàn cờ tổng quát kích thước n x n ô vuông (n chẵn và 6 ≤ n ≤ 20). Trên một ô nào đó có đặt một quân mã. Quân mã đang ở ô (X1, Y1) có thể di chuyển sang ô (X2, Y2) nếu |X1-X2|.|Y1-Y2| = 2 (Xem hình vẽ).



Hãy tìm một hành trình của quân mã từ ô xuất phát, đi qua tất cả các ô của bàn cờ, mỗi ô đúng 1 lần.

Ví dụ:

Với n = 8, ô xuất phát (3, 3)

45	42	3	18	35	20	5	8
2	17	44	41	4	7	34	21
43	46	1	36	19	50	9	6
16	31	48	59	40	33	22	51
47	60	37	32	49	58	39	10
30	15	64	57	38	25	52	23
61	56	13	28	63	54	11	26
14	29	62	55	12	27	24	53

Với n = 10, ô xuất phát (6, 5)

18	71	100	43	20	69	86	45	22	55
97	42	19	70	99	44	21	24	87	46
72	17	98	95	68	85	88	63	26	23
41	96	73	84	81	94	67	90	47	50
16	83	80	93	74	89	64	49	62	27
79	40	35	82	1	76	91	66	51	48
36	15	78	75	92	65	2	61	28	53
39	12	37	34	77	60	57	52	3	6
14	33	10	59	56	31	8	5	54	29
11	38	13	32	9	58	55	30	7	4

Gợi ý: Nếu coi các ô của bàn cờ là các đỉnh của đồ thị và các cạnh là nối giữa hai đỉnh tương ứng với hai ô mã giao chân thì dễ thấy rằng hành trình của quân mã cần tìm sẽ là một đường đi

Hamilton. Ta có thể xây dựng hành trình bằng thuật toán quay lui kết hợp với phương pháp duyệt ưu tiên Warnsdorff: Nếu gọi deg(x, y) là số ô kề với ô (x, y) và chưa đi qua (kề ở đây theo nghĩa đỉnh kề chứ không phải là ô kề cạnh) thì từ một ô ta sẽ **không thử xét lần lượt các hướng đi** có thể, mà ta sẽ **ưu tiên thử hướng đi tới ô có deg nhỏ nhất trước**. Trong **trường hợp có tồn tại đường đi**, phương pháp này hoạt động với tốc độ tuyệt vời: Với mọi n chẵn trong khoảng từ 6 tới 18, với mọi vị trí ô xuất phát, trung bình thời gian tính từ lúc bắt đầu tới lúc tìm ra một nghiệm < 1 giây. Tuy nhiên trong **trường hợp n lẻ, có lúc không tồn tại đường đi**, do phải duyệt hết mọi khả năng nên thời gian thực thi lại hết sức tồi tệ. (Có xét ưu tiên như trên hay xét thứ tự như trước kia thì cũng vậy thôi. Ta có thể thử với n lẻ: 5, 7, 9 … và ô xuất phát (1, 2), sau đó ngồi xem máy tính toát mồ hôi).

§8. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

8.1. ĐỒ THỊ CÓ TRỌNG SỐ

Đồ thị mà mỗi cạnh của nó được gán cho tương ứng với một số (nguyên hoặc thực) được gọi là đồ thị có trọng số. Số gán cho mỗi cạnh của đồ thị được gọi là trọng số của cạnh. Tương tự như đồ thị không trọng số, có nhiều cách biểu diễn đồ thị có trọng số trong máy tính. Đối với đơn đồ thị thì cách dễ dùng nhất là sử dụng ma trận trọng số:

Giả sử đồ thị G = (V, E) có n đỉnh. Ta sẽ dựng ma trận vuông C kích thước n x n. Ở đây:

Nếu (u, v) ∈ E thì C[u, v] = trọng số của cạnh (u, v)

Nếu (u, v) ∉ E thì tuỳ theo trường hợp cụ thể, C[u, v] được gán một giá trị nào đó để có thể nhận biết được (u, v) không phải là cạnh (Chẳng hạn có thể gán bằng +∞, hay bằng 0, bằng -∞ v.v…)

Quy ước c[v, v] = 0 với mọi đỉnh v.

Đường đi, chu trình trong đồ thị có trọng số cũng được định nghĩa giống như trong trường hợp không trọng số, chỉ có khác là độ dài đường đi không phải tính bằng số cạnh đi qua, mà được tính bằng tổng trọng số của các cạnh đi qua.

8.2. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

Trong các ứng dụng thực tế, chẳng hạn trong mạng lưới giao thông đường bộ, đường thuỷ hoặc đường không. Người ta không chỉ quan tâm đến việc tìm đường đi giữa hai địa điểm mà còn phải lựa chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn không gian, thời gian hay chi phí). Khi đó phát sinh yêu cầu tìm đường đi ngắn nhất giữa hai đỉnh của đồ thị. Bài toán đó phát biểu dưới dạng tổng quát như sau: Cho đồ thị có trọng số G = (V, E), hãy tìm một đường đi ngắn nhất từ đỉnh xuất phát S ∈ V đến đỉnh đích F ∈ V. Độ dài của đường đi này ta sẽ ký hiệu là d[S, F] và gọi là ***khoảng cách*** từ S đến F. Nếu như không tồn tại đường đi từ S tới F thì ta sẽ đặt khoảng cách đó = +∞.

Nếu như đồ thị có chu trình âm (chu trình với độ dài âm) thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định, bởi vì bằng cách đi vòng theo chu trình này một số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào. Trong trường hợp như vậy, có thể đặt vấn đề tìm **đường đi cơ bản** (đường đi không có đỉnh lặp lại) ngắn nhất. Vấn đề đó là một vấn đề hết sức phức tạp mà ta sẽ không bàn tới ở đây.

Nếu như đồ thị không có chu trình âm thì ta có thể chứng minh được rằng một trong những đường đi ngắn nhất là đường đi cơ bản. Và nếu như biết được khoảng cách từ S tới tất cả những đỉnh khác thì đường đi ngắn nhất từ S tới F có thể tìm được một cách dễ dàng qua thuật toán sau:

Gọi c[u, v] là trọng số của cạnh [u, v]. Qui ước c[v, v] = 0 với mọi $v \in V$ và c[u, v] = +∞ nếu như $(u, v) \notin E$. Đặt d[S, v] là khoảng cách từ S tới v. Để tìm đường đi từ S tới F, ta có thể nhận thấy rằng luôn tồn tại đỉnh $F_1 \neq F$ sao cho:

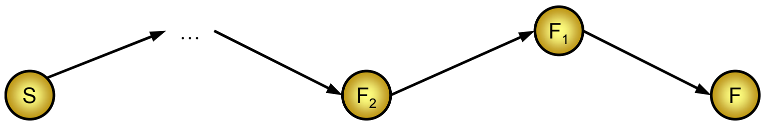
$$d[S, F] = d[S, F_1] + c[F_1, F]$$

(Độ dài đường đi ngắn nhất S->F = Độ dài đường đi ngắn nhất S->F_1 + Chi phí đi từ F_1 tới F)

Đỉnh F_1 đó là đỉnh liền trước F trong đường đi ngắn nhất từ S tới F. Nếu $F_1 \equiv S$ thì đường đi ngắn nhất là đường đi trực tiếp theo cung (S, F). Nếu không thì vấn đề trở thành tìm đường đi ngắn nhất từ S tới F_1. Và ta lại tìm được một đỉnh F_2 khác F và F_1 để:

$$d[S, F_1] = d[S, F_2] + c[F_2, F_1]$$

Cứ tiếp tục như vậy, sau một số hữu hạn bước, ta suy ra rằng dãy F, F_1, F_2, … không chứa đỉnh lặp lại và kết thúc ở S. Lật ngược thứ tự dãy cho ta đường đi ngắn nhất từ S tới F.



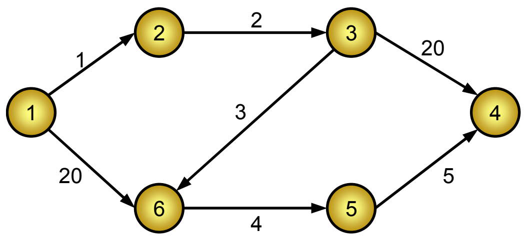
Tuy nhiên, người ta thường không sử dụng phương pháp này mà sẽ kết hợp lưu vết đường đi ngay trong quá trình tìm kiếm.

Dưới đây ta sẽ xét một số thuật toán tìm đường đi ngắn nhất từ đỉnh S tới đỉnh F trên đơn đồ thị có hướng G = (V, E) có n đỉnh và m cung. Trong trường hợp đơn đồ thị vô hướng với trọng số không âm, bài toán tìm đường đi ngắn nhất có thể dẫn về bài toán trên đồ thị có hướng bằng cách thay mỗi cạnh của nó bằng hai cung có hướng ngược chiều nhau. Lưu ý rằng các thuật toán dưới đây sẽ luôn luôn tìm được đường đi ngắn nhất là đường đi cơ bản.

Input: file văn bản MINPATH.INP

- Dòng 1: Chứa số đỉnh n (≤ 100), số cung m của đồ thị, đỉnh xuất phát S, đỉnh đích F cách nhau ít nhất 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, c[u, v] cách nhau ít nhất 1 dấu cách, thể hiện (u, v) là một cung $\in E$ và trọng số của cung đó là c[u,v] (c[u, v] là số nguyên có giá trị tuyệt đối ≤ 100)

Output: file văn bản MINPATH.OUT ghi đường đi ngắn nhất từ S tới F và độ dài đường đi đó



MINPATH.INP	MINPATH.OUT
6 7 1 4 1 2 1 1 6 20 2 3 2 3 6 3 3 4 20 5 4 5 6 5 4	Distance from 1 to 4: 15 4<-5<-6<-3<-2<-1

8.3. TRƯỜNG HỢP ĐỒ THỊ KHÔNG CÓ CHU TRÌNH ÂM - THUẬT TOÁN FORD BELLMAN

Thuật toán Ford-Bellman có thể phát biểu rất đơn giản:

Với đỉnh xuất phát S. Gọi d[v] là khoảng cách từ S tới v với các giá trị khởi tạo là:

- ***d[S] = 0***
- ***d[v] = +∞ nếu v ≠ S***

Sau đó ta tối ưu hoá dần các d[v] như sau: Xét mọi cặp đỉnh u, v của đồ thị, nếu có một cặp đỉnh u, v mà d[v] > d[u]+ c[u, v] thì ta đặt lại d[v] := d[u] + c[u, v]. Tức là nếu độ dài đường đi từ S tới v lại **lớn hơn** tổng độ dài đường đi từ S tới u cộng với chi phí đi từ u tới v thì ta sẽ huỷ bỏ đường đi từ S tới v đang có và coi đường đi từ S tới v chính là đường đi từ S tới u sau đó đi tiếp từ u tới v. Chú ý rằng ta đặt c[u, v] = +∞ nếu (u, v) không là cung. Thuật toán sẽ kết thúc khi không thể tối ưu thêm bất kỳ một nhãn d[v] nào nữa.

Tính dúng của thuật toán:

Tại bước khởi tạo thì mỗi d[v] chính là độ dài ngắn nhất của đường đi từ S tới v qua không quá 0 cạnh.

Giả sử khi bắt đầu bước lặp thứ i (i ≥ 1), d[v] đã bằng độ dài đường đi ngắn nhất từ S tới v qua không quá i - 1 cạnh. Do tính chất: đường đi từ S tới v qua không quá i cạnh sẽ phải thành lập bằng cách: lấy một đường đi từ S tới một đỉnh u nào đó qua không quá i - 1 cạnh, rồi đi tiếp tới v bằng cung (u, v), nên độ dài đường đi ngắn nhất từ S tới v qua không quá i cạnh sẽ được tính bằng giá trị nhỏ nhất trong các giá trị: (Nguyên lý tối ưu Bellman)

Độ dài đường đi ngắn nhất từ S tới v qua không quá i - 1 cạnh

Độ dài đường đi ngắn nhất từ S tới u qua không quá i - 1 cạnh cộng với trọng số cạnh (u, v) (∀u)

Vì vậy, sau bước lặp tối ưu các d[v] bằng công thức

$$d[v]_{\text{bước } i} = \min(d[v]_{\text{bước } i-1}, d[u]_{\text{bước } i-1} + c[u, v]) \ (\forall u)$$

thì các d[v] sẽ bằng độ dài đường đi ngắn nhất từ S tới v qua không quá i cạnh.

Sau bước lặp tối ưu thứ n - 1, ta có d[v] = độ dài đường đi ngắn nhất từ S tới v qua không quá n - 1 cạnh. Vì đồ thị không có chu trình âm nên sẽ có một đường đi ngắn nhất từ S tới v là đường đi cơ bản (qua không quá n - 1 cạnh). Tức là d[v] sẽ là độ dài đường đi ngắn nhất từ S tới v.

Vậy thì số bước lặp tối ưu hoá sẽ không quá n - 1 bước.

Trong khi cài đặt chương trình, nếu mỗi bước lặp được mô tả dưới dạng:

```
for u := 1 to n do
  for v := 1 to n do
    d[v] := min(d[v], d[u] + c[u, v]);
```

Do sự tối ưu bắc cầu (dùng d[u] tối ưu d[v] rồi lại có thể dùng d[v] tối ưu d[w] nữa…) chỉ làm tốc độ tối ưu nhãn d[v] tăng nhanh hơn nên số bước lặp tối ưu nhãn vẫn sẽ không quá n - 1 bước

P_4_08_1.PAS * Thuật toán Ford-Bellman

```
program Shortest_Path_by_Ford_Bellman;
const
  InputFile  = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Trace: array[1..max] of Integer;
  n, S, F: Integer;

procedure LoadGraph; {Nhập đồ thị, đồ thị không được có chu trình âm}
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, S, F);
  {Những cạnh không có trong đồ thị được gán trọng số +∞}
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure Init; {Khởi tạo}
var
  i: Integer;
begin
  for i := 1 to n do d[i] := MaxC;
  d[S] := 0;
end;

procedure Ford_Bellman; {Thuật toán Ford-Bellman}
var
  Stop: Boolean;
  u, v, CountLoop: Integer;
begin
  for CountLoop := 1 to n - 1 do
    begin
      Stop := True;
      for u := 1 to n do
        for v := 1 to n do
          if d[v] > d[u] + c[u, v] then   {Nếu ∃u, v thoả mãn d[v] > d[u] + c[u, v] thì tối ưu lại d[v]}
            begin
              d[v] := d[u] + c[u, v];
              Trace[v] := u; {Lưu vết đường đi}
              Stop := False;
            end;
      if Stop then Break;
    end;
```

```
  {Thuật toán kết thúc khi không sửa nhãn các d[v] được nữa hoặc đã lặp đủ n - 1 lần }
end;

procedure PrintResult; {In đường đi từ S tới F}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[F] = maxC then   {Nếu d[F] vẫn là +∞ thì tức là không có đường}
    WriteLn(fo, 'Path from ', S, ' to ', F, ' not found')
  else  {Truy vết tìm đường đi}
    begin
      WriteLn(fo, 'Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
  Close(fo);
end;

begin
  LoadGraph;
  Init;
  Ford_Bellman;
  PrintResult;
end.
```

8.4. TRƯỜNG HỢP TRỌNG SỐ TRÊN CÁC CUNG KHÔNG ÂM - THUẬT TOÁN DIJKSTRA

Trong trường hợp trọng số trên các cung không âm, thuật toán do Dijkstra đề xuất dưới đây hoạt động hiệu quả hơn nhiều so với thuật toán Ford-Bellman. Ta hãy xem trong trường hợp này, thuật toán Ford-Bellman thiếu hiệu quả ở chỗ nào:

Với đỉnh $v \in V$, Gọi d[v] là độ dài đường đi ngắn nhất từ S tới v. Thuật toán Ford-Bellman khởi gán d[S] = 0 và $d[v] = +\infty$ với $\forall v \neq S$, sau đó tối ưu hoá dần các nhãn d[v] bằng cách sửa nhãn theo công thức: d[v] := min(d[v], d[u] + c[u, v]) với $\forall u, v \in V$. Như vậy nếu như ta dùng đỉnh u sửa nhãn đỉnh v, sau đó nếu ta lại tối ưu được d[u] thêm nữa thì ta cũng phải sửa lại nhãn d[v] dẫn tới việc d[v] có thể phải chỉnh đi chỉnh lại rất nhiều lần. Vậy nên chăng, tại mỗi bước **không phải ta xét mọi cặp đỉnh (u, v)** để dùng đỉnh u sửa nhãn đỉnh v mà sẽ **chọn đỉnh u là đỉnh mà không thể tối ưu nhãn d[u]** thêm được nữa.

Thuật toán Dijkstra (E.Dijkstra - 1959) có thể mô tả như sau:

Bước 1: Khởi tạo

Với đỉnh $v \in V$, gọi nhãn d[v] là độ dài đường đi ngắn nhất từ S tới v. Ta sẽ tính các d[v]. Ban đầu d[v] được khởi gán như trong thuật toán Ford-Bellman (d[S] = 0 và $d[v] = \infty$ với $\forall v \neq S$). Nhãn của mỗi đỉnh có hai trạng thái tự do hay cố định, nhãn tự do có nghĩa là có thể còn tối ưu hơn được nữa và nhãn cố định tức là d[v] đã bằng độ dài đường đi ngắn nhất từ S tới v nên không thể tối ưu thêm.

Để làm điều này ta có thể sử dụng kỹ thuật đánh dấu: Free[v] = TRUE hay FALSE tuỳ theo d[v] tự do hay cố định. Ban đầu các nhãn đều tự do.

Bước 2: Lặp

Bước lặp gồm có hai thao tác:

1. **Cố định nhãn: Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh u là đỉnh có d[u] nhỏ nhất, và cố định nhãn đỉnh u**.
2. **Sửa nhãn: Dùng đỉnh u, xét tất cả những đỉnh v và sửa lại các d[v] theo công thức:**

d[v] := min(d[v], d[u] + c[u, v])

Bước lặp sẽ kết thúc khi mà đỉnh đích F được cố định nhãn (tìm được đường đi ngắn nhất từ S tới F); hoặc tại thao tác cố định nhãn, tất cả các đỉnh tự do đều có nhãn là +∞ (không tồn tại đường đi).

Có thể đặt câu hỏi, ở thao tác 1, tại sao đỉnh u như vậy được cố định nhãn, giả sử d[u] còn có thể tối ưu thêm được nữa thì tất phải có một đỉnh t mang nhãn tự do sao cho d[u] > d[t] + c[t, u]. Do trọng số c[t, u] không âm nên d[u] > d[t], trái với cách chọn d[u] là nhỏ nhất. Tất nhiên trong lần lặp đầu tiên thì S là đỉnh được cố định nhãn do d[S] = 0.

Bước 3: Kết hợp với việc lưu vết đường đi trên từng bước sửa nhãn, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi (d[F] = +∞).

P_4_08_2.PAS * Thuật toán Dijkstra

```
program Shortest_Path_by_Dijkstra;
const
  InputFile  = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Trace: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  n, S, F: Integer;

procedure LoadGraph; {Nhập đồ thị, trọng số các cung phải là số không âm}
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, S, F);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure Init; {Khởi tạo các nhãn d[v], các đỉnh đều được coi là tự do}
var
  i: Integer;
begin
```

```
  for i := 1 to n do d[i] := MaxC;
  d[S] := 0;
  FillChar(Free, SizeOf(Free), True);
end;

procedure Dijkstra; {Thuật toán Dijkstra}
var
  i, u, v: Integer;
  min: Integer;
begin
  repeat
 {Tìm trong các đỉnh có nhãn tự do ra đỉnh u có d[u] nhỏ nhất}
    u := 0; min := maxC;
    for i := 1 to n do
      if Free[i] and (d[i] < min) then
        begin
          min := d[i];
          u := i;
        end;
    {Thuật toán sẽ kết thúc khi các đỉnh  tự do đều có nhãn +∞ hoặc đã chọn đến đỉnh F}
    if (u = 0) or (u = F) then Break;
    {Cố định nhãn đỉnh u}
    Free[u] := False;
    {Dùng đỉnh u tối ưu nhãn những đỉnh tự do kề với u}
    for v := 1 to n do
      if Free[v] and (d[v] > d[u] + c[u, v]) then
        begin
          d[v] := d[u] + c[u, v];
          Trace[v] := u;
        end;
  until False;
end;

procedure PrintResult; {In đường đi từ S tới F}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[F] = maxC then
    WriteLn(fo, 'Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn(fo, 'Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
  Close(fo);
end;

begin
  LoadGraph;
  Init;
  Dijkstra;
  PrintResult;
end.
```

8.5. THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC HEAP

Nếu đồ thị có nhiều đỉnh, ít cạnh, ta có thể sử dụng danh sách kề kèm trọng số để biểu diễn đồ thị, tuy nhiên tốc độ của thuật toán DIJKSTRA vẫn khá chậm vì trong trường hợp xấu nhất, nó cần n lần cố định nhãn và mỗi lần tìm đỉnh để cố định nhãn sẽ mất một đoạn chương trình với độ phức tạp O(n). Để tăng tốc độ, người ta thường sử dụng cấu trúc dữ liệu Heap để lưu các đỉnh chưa cố định nhãn. Heap ở đây là một cây nhị phân hoàn chỉnh thoả mãn: Nếu u là đỉnh lưu ở nút cha và v là đỉnh lưu ở nút con thì d[u] ≤ d[v]. (Đỉnh r lưu ở gốc Heap là đỉnh có d[r] nhỏ nhất).

Tại mỗi bước lặp của thuật toán Dijkstra có hai thao tác: Tìm đỉnh cố định nhãn và Sửa nhãn.

Với mỗi đỉnh v, gọi Pos[v] là vị trí đỉnh v trong Heap, quy ước Pos[v] = 0 nếu v chưa bị đẩy vào Heap. Mỗi lần có thao tác sửa đổi vị trí các đỉnh trên cấu trúc Heap, ta lưu ý cập nhập lại mảng Pos này.

Thao tác tìm đỉnh cố định nhãn sẽ lấy đỉnh lưu ở gốc Heap, cố định nhãn, đưa phần tử cuối Heap vào thế chỗ và thực hiện việc vun đống (Adjust).

Thao tác sửa nhãn, sẽ duyệt danh sách kề của đỉnh vừa cố định nhãn và sửa nhãn những đỉnh tự do kề với đỉnh này, mỗi lần sửa nhãn một đỉnh nào đó, ta xác định đỉnh này nằm ở đâu trong Heap (dựa vào mảng Pos) và thực hiện việc chuyển đỉnh đó lên (UpHeap) phía gốc Heap nếu cần để bảo toàn cấu trúc Heap.

Cài đặt dưới đây có Input/Output giống như trên nhưng có thể thực hiện trên đồ thị 5000 đỉnh, 10000 cạnh, trọng số mỗi cạnh ≤ 10000.

P_4_08_3.PAS * Thuật toán Dijkstra và cấu trúc Heap

```
program Shortest_Path_by_Dijkstra_and_Heap;
const
  InputFile  = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 5000;
  maxE = 10000;
  maxC = 1000000000;
type
  TAdj = array[1..maxE] of Integer;
  TAdjCost = array[1..maxE] of LongInt;
  THeader = array[1..max + 1] of Integer;
var
  adj: ^TAdj; {Danh sách kề dạng mảng}
  adjCost: ^TAdjCost; {Kèm trọng số}
  h: ^THeader; {Mảng đánh dấu các đoạn trong mảng adj^ chứa danh sách kề}
  d: array[1..max] of LongInt;
  Trace: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  heap: array[1..max] of Integer; {heap[i] = đỉnh lưu tại nút i của heap}
  Pos: array[1..max] of Integer; {pos[v] = vị trí của nút v trong heap (tức là pos[heap[i]] = i)}
  n, S, F, nHeap: Integer;

procedure LoadGraph; {Nhập dữ liệu}
var
  i, m, u, v, c: Integer;
  fi: Text;
begin
```

```
  {Đọc file lần 1, để xác định các đoạn}
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, S, F);
  New(h);
  New(adj); New(adjCost);
  {Phép đếm phân phối (Distribution Counting)}
  FillChar(h^, SizeOf(h^), 0);
  for i := 1 to m do
    begin
      ReadLn(fi, u); {Ta chỉ cần tính bán bậc ra (deg+) của mỗi đỉnh nên không cần đọc đủ 3 thành phần}
      Inc(h^[u]);
    end;
  for i := 2 to n do h^[i] := h^[i - 1] + h^[i];
  Close(fi);
  {Đến đây, ta xác định được h[u] là vị trí cuối của danh sách kề đỉnh u trong adj^}
  Reset(fi); {Đọc file lần 2, vào cấu trúc danh sách kề}
  ReadLn(fi); {Bỏ qua dòng đầu tiên Input file}
  for i := 1 to m do
    begin
      ReadLn(fi, u, v, c);
      adj^[h^[u]] := v; {Điền v và c vào vị trí đúng trong danh sách kề của u}
      adjCost^[h^[u]] := c;
      Dec(h^[u]);
    end;
  h^[n + 1] := m;
  Close(fi);
end;

procedure Init; {Khởi tạo d[i] = độ dài đường đi ngắn nhất từ S tới i qua 0 cạnh, Heap rỗng}
var
  i: Integer;
begin
  for i := 1 to n do d[i] := maxC;
  d[S] := 0;
  FillChar(Free, SizeOf(Free), True);
  FillChar(Pos, SizeOf(Pos), 0);
  nHeap := 0;
end;

procedure Update(v: Integer); {Đỉnh v vừa được sửa nhãn, cần phải chỉnh lại Heap}
var
  parent, child: Integer;
begin
  child := Pos[v]; {child là vị trí của v trong Heap}
  if child = 0 then {Nếu v chưa có trong Heap thì Heap phải bổ sung thêm 1 phần tử và coi child = nút lá cuối Heap}
    begin
      Inc(nHeap); child := nHeap;
    end;
  parent := child div 2; {parent là nút cha của child}
  while (parent > 0) and (d[heap[parent]] > d[v]) do
    begin {Nếu đỉnh lưu ở nút parent ưu tiên kém hơn v thì đỉnh đó sẽ bị đẩy xuống nút con child}
      heap[child] := heap[parent]; {Đẩy đỉnh lưu trong nút cha xuống nút con}
      Pos[heap[child]] := child; {Ghi nhận lại vị trí mới của đỉnh đó}
      child := parent; {Tiếp tục xét lên phía nút gốc}
      parent := child div 2;
    end;
  {Thao tác "kéo xuống"  ở trên tạo ra một "khoảng trống" tại nút child của Heap, đỉnh v sẽ được đặt vào đây}
  heap[child] := v;
  Pos[v] := child;
end;

function Pop: Integer;
var
  r, c, v: Integer;
```

```
begin
  Pop := heap[1]; {Nút gốc Heap chứa đỉnh có nhãn tự do nhỏ nhất}
  v := heap[nHeap]; {v là đỉnh ở nút lá cuối Heap, sẽ được đảo lên đầu và vun đống}
  Dec(nHeap);
  r := 1; {Bắt đầu từ nút gốc}
  while r * 2 <= nHeap do {Chừng nào r chưa phải là lá}
    begin
   {Chọn c là nút chứa đỉnh ưu tiên hơn trong hai nút con}
      c := r * 2;
      if (c < nHeap) and (d[heap[c + 1]] < d[heap[c]]) then Inc(c);
   {Nếu v ưu tiên hơn cả đỉnh chứa trong C, thì thoát ngay}
      if d[v] <= d[heap[c]] then Break;
      heap[r] := heap[c]; {Chuyển đỉnh lưu ở nút con c lên nút cha r}
      Pos[heap[r]] := r; {Ghi nhận lại vị trí mới trong Heap của đỉnh đó}
      r := c; {Gán nút cha := nút con và lặp lại}
    end;
  heap[r] := v; {Đỉnh v sẽ được đặt vào nút r để bảo toàn cấu trúc Heap}
  Pos[v] := r;
end;

procedure Dijkstra;
var
  i, u, iv, v, min: Integer;
begin
  Update(1);
  repeat
    u := Pop; {Chọn đỉnh tự do có nhãn nhỏ nhất}
    if u = F then Break; {Nếu đỉnh đó là F thì dừng ngay}
    Free[u] := False; {Cố định nhãn đỉnh đó}
    for iv := h^[u] + 1 to h^[u + 1] do {Xét danh sách kề}
      begin
        v := adj^[iv];
        if Free[v] and (d[v] > d[u] + adjCost^[iv]) then
          begin
            d[v] := d[u] + adjCost^[iv]; {Tối ưu hoá nhãn của các đỉnh tự do kề với u}
            Trace[v] := u; {Lưu vết đường đi}
            Update(v); {Tổ chức lại Heap}
          end;
      end;
  until nHeap = 0; {Không còn đỉnh nào mang nhãn tự do}
end;

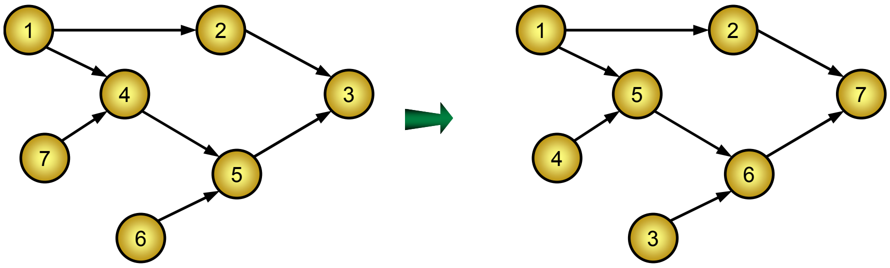
procedure PrintResult;
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[F] = maxC then
    WriteLn(fo, 'Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn(fo, 'Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
  Close(fo);
end;

begin
  LoadGraph;
```

```
  Init;
  Dijkstra;
  PrintResult;
end.
```

8.6. TRƯỜNG HỢP ĐỒ THỊ KHÔNG CÓ CHU TRÌNH - THỨ TỰ TÔ PÔ

Ta có định lý sau: Giả sử G = (V, E) là đồ thị không có chu trình (có hướng - tất nhiên). Khi đó các đỉnh của nó có thể đánh số sao cho mỗi cung của nó chỉ nối từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn.



Hình 76: Phép đánh lại chỉ số theo thứ tự tôpô

Thuật toán đánh số lại các đỉnh của đồ thị có thể mô tả như sau:

Trước hết ta chọn một đỉnh không có cung đi vào và đánh chỉ số 1 cho đỉnh đó. Sau đó xoá bỏ đỉnh này cùng với tất cả những cung từ u đi ra, ta được một đồ thị mới cũng không có chu trình, và lại đánh chỉ số 2 cho một đỉnh v nào đó không có cung đi vào, rồi lại xoá đỉnh v cùng với các cung từ v đi ra … Thuật toán sẽ kết thúc nếu như hoặc ta đã đánh chỉ số được hết các đỉnh, hoặc tất cả các đỉnh còn lại đều có cung đi vào. Trong trường hợp tất cả các đỉnh còn lại đều có cung đi vào thì sẽ tồn tại chu trình trong đồ thị và khẳng định thuật toán tìm đường đi ngắn nhất trong mục này không áp dụng được. (Thuật toán đánh số này có thể cải tiến bằng cách dùng một hàng đợi và cho những đỉnh không có cung đi vào đứng chờ lần lượt trong hàng đợi đó, lần lượt rút các đỉnh khỏi hàng đợi và đánh số cho nó, đồng thời huỷ những cung đi ra khỏi đỉnh vừa đánh số, lưu ý sau mỗi lần loại bỏ cung (u, v), nếu thấy bán bậc vào của v = 0 thì đẩy v vào chờ trong hàng đợi, như vậy đỡ mất công duyệt để tìm những đỉnh có bán bậc vào = 0)

Nếu các đỉnh được đánh số sao cho mỗi cung phải nối từ một đỉnh tới một đỉnh khác mang chỉ số lớn hơn thì thuật toán tìm đường đi ngắn nhất có thể mô tả rất đơn giản:

Gọi d[v] là độ dài đường đi ngắn nhất từ S tới v. Khởi tạo d[S] = 0 và d[v] = ∞ với $\forall v \neq S$. Ta sẽ tính các d[v] như sau:

```
for u := 1 to n - 1 do
  for v := u + 1 to n do
     d[v] := min(d[v], d[u] + c[u, v]);
```

(Giả thiết rằng c[u, v] = $+\infty$ nếu như (u, v) không là cung).

Tức là dùng đỉnh u, tối ưu nhãn d[v] của những đỉnh v nối từ u, với u được xét lần lượt từ 1 tới n - 1. Có thể làm tốt hơn nữa bằng cách chỉ cần cho u chạy từ đỉnh xuất phát S tới đỉnh kết thúc F. Bởi hễ **u chạy tới đâu thì nhãn d[u] là không thể cực tiểu hoá thêm nữa.**

P_4_08_4.PAS * Đường đi ngắn nhất trên đồ thị không có chu trình

```
program Critical_Path;
const
  InputFile  = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  List, d, Trace: array[1..max] of Integer; {List là danh sách các đỉnh theo cách đánh số mới}
  n, S, F, count: Integer;

procedure LoadGraph; {Nhập dữ liệu, đồ thị không được có chu trình}
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, S, F);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure Number; {Thuật toán đánh số các đỉnh}
var
  deg: array[1..max] of Integer;
  u, v: Integer;
  front: Integer;
begin
  {Trước hết, tính bán bậc vào của các đỉnh (deg-)}
  FillChar(deg, SizeOf(deg), 0);
  for u := 1 to n do
    for v := 1 to n do
      if (v <> u) and (c[v, u] < maxC) then Inc(deg[u]);
  {Đưa những đỉnh có bán bậc vào = 0 vào danh sách List}
  count := 0;
  for u := 1 to n do
    if deg[u] = 0 then
      begin
        Inc(count); List[count] := u;
      end;
  {front: Chỉ số phần tử đang xét, count: Số phần tử trong danh sách}
  front := 1;
  while front <= count do {Chừng nào chưa xét hết các phần tử trong danh sách}
    begin
      {Xét phần tử thứ front trong danh sách, đẩy con trỏ front sang phần tử kế tiếp}
      u := List[front]; Inc(front);
      for v := 1 to n do
        if c[u, v] <> maxC then {Xét những cung (u, v) và "loại" khỏi đồ thị ⇔ deg-(v) giảm 1}
          begin
            Dec(deg[v]);
            if deg[v] = 0 then {Nếu v trở thành đỉnh không có cung đi vào}
              begin {Đưa tiếp v vào danh sách List}
                Inc(count);
                List[count] := v;
              end;
          end;
    end;
end;
```

```
procedure Init;
var
  i: Integer;
begin
  for i := 1 to n do d[i] := maxC;
  d[S] := 0;
end;

procedure FindPath; {Thuật toán quy hoạch động tìm đường đi ngắn nhất trên đồ thị không chu trình}
var
  i, j, u, v: Integer;
begin
  for i := 1 to n - 1 do
    for j := i + 1 to n do
      begin
        u := List[i]; v := List[j]; {Dùng List[i] tối ưu nhãn List[j] với i < j}
        if d[v] > d[u] + c[u, v] then
          begin
            d[v] := d[u] + c[u, v];
            Trace[v] := u;
          end
      end;
end;

procedure PrintResult; {In đường đi từ S tới F}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[F] = maxC then
    WriteLn(fo, 'Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn(fo, 'Distance from ', S, ' to ', F, ': ', d[F]);
      while F <> S do
        begin
          Write(fo, F, '<-');
          F := Trace[F];
        end;
      WriteLn(fo, S);
    end;
  Close(fo);
end;

begin
  LoadGraph;
  Number;
  Init;
  FindPath;
  PrintResult;
end.
```

8.7. ĐƯỜNG ĐI NGẮN NHẤT GIỮA MỌI CẶP ĐỈNH - THUẬT TOÁN FLOYD

Cho đơn đồ thị có hướng, có trọng số G = (V, E) với n đỉnh và m cạnh. Bài toán đặt ra là hãy tính tất cả các d(u, v) là khoảng cách từ u tới v. Rõ ràng là ta có thể áp dụng thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với n khả năng chọn đỉnh xuất phát. Nhưng ta có cách làm gọn hơn

nhiều, cách làm này rất giống với thuật toán Warshall mà ta đã biết: Từ ma trận trọng số c, thuật toán Floyd tính lại các c[u, v] thành độ dài đường đi ngắn nhất từ u tới v:

Với mọi đỉnh k của đồ thị được xét theo thứ tự từ 1 tới n, xét mọi cặp đỉnh u, v. Cực tiểu hoá c[u, v] theo công thức:

$$c[u, v] := \min(c[u, v], c[u, k] + c[k, v])$$

Tức là nếu như đường đi từ u tới v đang có lại dài hơn đường đi từ u tới k cộng với đường đi từ k tới v thì ta huỷ bỏ đường đi từ u tới v hiện thời và coi đường đi từ u tới v sẽ là nối của hai đường đi từ u tới k rồi từ k tới v (Chú ý rằng ta còn có việc lưu lại vết):

```
for k := 1 to n do
    for u := 1 to n do
        for v := 1 to n do
            c[u, v] := min(c[u, v], c[u, k] + c[k, v]);
```

Tính đúng của thuật toán:

Gọi ck[u, v] là độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập {1, 2, …, k}. Rõ ràng khi k = 0 thì $c^0[u, v] = c[u, v]$ (đường đi ngắn nhất là đường đi trực tiếp).

Giả sử ta đã tính được các $c^{k-1}[u, v]$ thì $c^k[u, v]$ sẽ được xây dựng như sau:

Nếu đường đi ngắn nhất từ u tới v mà chỉ qua các đỉnh trung gian thuộc tập {1, 2, …, k} lại:

- Không đi qua đỉnh k thì tức là chỉ qua các đỉnh trung gian thuộc tập {1, 2, …, k - 1} thì

$$c^k[u, v] = c^{k-1}[u, v]$$

  - Có đi qua đỉnh k thì đường đi đó sẽ là nối của một đường đi từ u tới k và một đường đi từ k tới v, hai đường đi này chỉ đi qua các đỉnh trung gian thuộc tập {1, 2, …, k - 1}.

$$c^k[u, v] = c^{k-1}[u, k] + c^{k-1}[k, v].$$

Vì ta muốn $c^k[u, v]$ là cực tiểu nên suy ra: $c^k[u, v] = \min(c^{k-1}[u, v], c^{k-1}[u, k] + c^{k-1}[k, v])$.

Và cuối cùng, ta quan tâm tới $c^n[u, v]$: Độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập {1, 2, ..., n}.

Khi cài đặt, thì ta sẽ không có các khái niệm $c^k[u, v]$ mà sẽ thao tác trực tiếp trên các trọng số c[u, v]. c[u, v] tại bước tối ưu thứ k sẽ được tính toán để tối ưu qua các giá trị c[u, v]; c[u, k] và c[k, v] tại bước thứ k - 1. Tính chính xác của cách cài đặt dưới dạng ba vòng lặp for lồng như trên có thể thấy được do sự tối ưu bắc cầu chỉ làm tăng tốc độ tối ưu các c[u, v] trong mỗi bước

P_4_08_5.PAS * Thuật toán Floyd

```
program Shortest_Path_by_Floyd;
const
  InputFile  = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 100;
  maxC = 10000;
var
  c: array[1..max, 1..max] of Integer;
  Trace: array[1..max, 1..max] of Integer; {Trace[u, v] = Đỉnh liền sau u trên đường đi từ u tới v}
  n, S, F: Integer;

procedure LoadGraph; {Nhập dữ liệu, đồ thị không được có chu trình âm}
```

```
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, S, F);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure Floyd;
var
  k, u, v: Integer;
begin
  for u := 1 to n do
    for v := 1 to n do Trace[u, v] := v; {Giả sử đường đi ngắn nhất giữa mọi cặp đỉnh là đường trực tiếp}
  {Thuật toán Floyd}
  for k := 1 to n do
    for u := 1 to n do
      for v := 1 to n do
        if c[u, v] > c[u, k] + c[k, v] then {Đường đi từ qua k tốt hơn}
          begin
            c[u, v] := c[u, k] + c[k, v]; {Ghi nhận đường đi đó thay cho đường cũ}
            Trace[u, v] := Trace[u, k]; {Lưu vết đường đi}
          end;
end;

procedure PrintResult; {In đường đi từ S tới F}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if c[S, F] = maxC
    then WriteLn(fo, 'Path from ', S, ' to ', F, ' not found')
  else
    begin
      WriteLn(fo, 'Distance from ', S, ' to ', F, ': ', c[S, F]);
      repeat
        Write(fo, S, '->');
        S := Trace[S, F]; {Nhắc lại rằng Trace[S, F] là đỉnh liền sau S trên đường đi tới F}
      until S = F;
      WriteLn(fo, F);
    end;
  Close(fo);
end;

begin
  LoadGraph;
  Floyd;
  PrintResult;
end.
```

Khác biệt rõ ràng của thuật toán Floyd là khi cần tìm đường đi ngắn nhất giữa một cặp đỉnh khác, chương trình chỉ việc in kết quả chứ không phải thực hiện lại thuật toán Floyd nữa.

8.8. NHẬN XÉT

Bài toán đường đi dài nhất trên đồ thị trong một số trường hợp có thể giải quyết bằng cách đổi dấu trọng số tất cả các cung rồi tìm đường đi ngắn nhất, nhưng hãy cẩn thận, có thể xảy ra trường hợp có chu trình âm.

Trong tất cả các cài đặt trên, vì sử dụng ma trận trọng số chứ không sử dụng danh sách cạnh hay danh sách kề có trọng số, nên ta đều đưa về đồ thị đầy đủ và **đem trọng số +∞ gán cho những cạnh không có trong đồ thị ban đầu**. Trên máy tính thì không có khái niệm trừu tượng +∞ nên ta sẽ phải chọn một số dương đủ lớn để thay. Như thế nào là đủ lớn? **số đó phải đủ lớn hơn tất cả trọng số của các đường đi cơ bản** để cho dù đường đi thật có tồi tệ đến đâu vẫn tốt hơn đường đi trực tiếp theo cạnh tưởng tượng ra đó. Vậy nên nếu đồ thị cho số đỉnh cũng như trọng số các cạnh vào cỡ 300 chẳng hạn thì giá trị đó **không thể chọn trong phạm vi Integer hay Word.** Ma trận c sẽ phải khai báo là **ma trận LongInt** và giá trị hằng số maxC trong các chương trình trên phải đổi lại là 300 * 299 + 1 - điều đó có thể gây ra nhiều phiền toái, chẳng hạn như vấn đề lãng phí bộ nhớ. Để khắc phục, người ta có thể cài đặt bằng danh sách kề kèm trọng số hoặc sử dụng những kỹ thuật đánh dấu khéo léo trong từng trường hợp cụ thể. Tuy nhiên có một điều chắc chắn: khi đồ thị cho số đỉnh cũng như trọng số các cạnh vào khoảng 300 thì các **trọng số c[u, v] trong thuật toán Floyd** và các **nhãn d[v] trong ba thuật toán còn lại** chắc chắn **không thể khai báo là Integer** được.

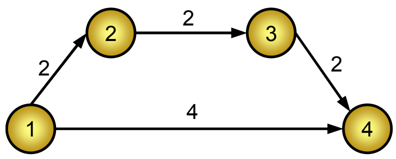
Xét về độ phức tạp tính toán, nếu cài đặt như trên, thuật toán Ford-Bellman có độ phức tạp là $O(n^3)$, thuật toán Dijkstra là $O(n^2)$, thuật toán tối ưu nhãn theo thứ tự tôpô là $O(n^2)$ còn thuật toán Floyd là $O(n^3)$. Tuy nhiên nếu sử dụng danh sách kề, thuật toán tối ưu nhãn theo thứ tự tôpô sẽ có độ phức tạp tính toán là O(m). Thuật toán Dijkstra kết hợp với cấu trúc dữ liệu Heap có độ phức tạp O(max(n, m).logn).

Khác với một bài toán đại số hay hình học có nhiều cách giải thì chỉ cần nắm vững một cách cũng có thể coi là đạt yêu cầu, những thuật toán tìm đường đi ngắn nhất bộc lộ rất rõ ưu, nhược điểm trong từng trường hợp cụ thể (Ví dụ như số đỉnh của đồ thị quá lớn làm cho không thể biểu diễn bằng ma trận trọng số thì thuật toán Floyd sẽ gặp khó khăn, hay thuật toán Ford-Bellman làm việc khá chậm). Vì vậy yêu cầu trước tiên là phải hiểu bản chất và thành thạo trong việc cài đặt tất cả các thuật toán trên để có thể sử dụng chúng một cách uyển chuyển trong từng trường hợp cụ thể. Những bài tập sau đây cho ta thấy rõ điều đó.

Bài tập

Bài 1

Giải thích tại sao đối với đồ thị sau, cần tìm đường đi dài nhất từ đỉnh 1 tới đỉnh 4 lại không thể dùng thuật toán Dijkstra được, cứ thử áp dụng thuật toán Dijkstra theo từng bước xem sao:



Bài 2

Trên mặt phẳng cho n đường tròn (n ≤ 2000), đường tròn thứ i được cho bởi bộ ba số thực (X_i, Y_i, R_i), (X_i, Y_i) là toạ độ tâm và R_i là bán kính. Chi phí di chuyển trên mỗi đường tròn bằng 0. Chi phí di chuyển giữa hai đường tròn bằng khoảng cách giữa chúng. Hãy tìm phương án di chuyển giữa hai đường tròn S, F cho trước với chi phí ít nhất.

Bài 3

Cho một dãy n số nguyên A[1], A[2], …, A[n] (n ≤ 10000; 1 ≤ A[i] ≤ 10000). Hãy tìm một dãy con gồm nhiều nhất các phần tử của dãy đã cho mà tổng của hai phần tử liên tiếp là số nguyên tố.

Bài 4

Một công trình lớn được chia làm n công đoạn đánh số 1, 2, …, n. Công đoạn i phải thực hiện mất thời gian t[i]. Quan hệ giữa các công đoạn được cho bởi bảng a[i, j]: a[i, j] = TRUE ⇔ công đoạn j chỉ được bắt đầu khi mà công việc i đã xong. Hai công đoạn độc lập nhau có thể tiến hành song song, hãy bố trí lịch thực hiện các công đoạn sao cho thời gian hoàn thành cả công trình là sớm nhất, cho biết thời gian sớm nhất đó.

Bài 5

Cho một bảng các số tự nhiên kích thước mxn (1 ≤ m, n ≤ 100). Từ một ô có thể di chuyển sang một ô kề cạnh với nó. Hãy tìm một cách đi từ ô (x, y) ra một ô biên sao cho tổng các số ghi trên các ô đi qua là cực tiểu.

§9. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

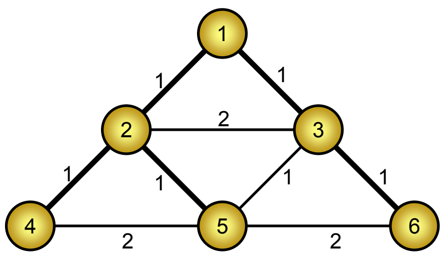
9.1. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

Cho G = (V, E) là đồ thị vô hướng liên thông có trọng số, với một cây khung T của G, ta gọi trọng số của cây T là tổng trọng số các cạnh trong T. Bài toán đặt ra là trong số các cây khung của G, chỉ ra cây khung có trọng số nhỏ nhất, cây khung như vậy được gọi là cây khung (hay cây bao trùm) nhỏ nhất của đồ thị, và bài toán đó gọi là bài toán cây khung nhỏ nhất. Sau đây ta sẽ xét hai thuật toán thông dụng để giải bài toán cây khung nhỏ nhất của đơn đồ thị vô hướng có trọng số.

Input: file văn bản MINTREE.INP:

- Dòng 1: Ghi hai số số đỉnh n (≤ 100) và số cạnh m của đồ thị cách nhau ít nhất 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng 3 số u, v, c[u, v] cách nhau ít nhất 1 dấu cách thể hiện đồ thị có cạnh (u, v) và trọng số cạnh đó là c[u, v]. (c[u, v] là số nguyên có giá trị tuyệt đối không quá 100).

Output: file văn bản MINTREE.OUT ghi các cạnh thuộc cây khung và trọng số cây khung



MINTREE.INP	MINTREE.OUT
6 9 1 2 1 1 3 1 2 4 1 2 3 2 2 5 1 3 5 1 3 6 1 4 5 2 5 6 2	Minimal spanning tree: (2, 4) = 1 (3, 6) = 1 (2, 5) = 1 (1, 3) = 1 (1, 2) = 1 Weight = 5

9.2. THUẬT TOÁN KRUSKAL (JOSEPH KRUSKAL - 1956)

Thuật toán Kruskal dựa trên mô hình xây dựng cây khung bằng thuật toán hợp nhất (§5), chỉ có điều thuật toán không phải xét các cạnh với thứ tự tuỳ ý mà xét các cạnh theo thứ tự đã sắp xếp: Với đồ thị vô hướng G = (V, E) có n đỉnh. Khởi tạo cây T ban đầu không có cạnh nào. Xét tất cả các cạnh của đồ thị **từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn**, nếu việc thêm cạnh đó vào T **không tạo thành chu trình đơn** trong T thì **kết nạp thêm** cạnh đó vào T. Cứ làm như vậy cho tới khi:

Hoặc đã kết nạp được n - 1 cạnh vào trong T thì ta được T là cây khung nhỏ nhất

Hoặc chưa kết nạp đủ n - 1 cạnh nhưng hễ cứ kết nạp thêm một cạnh bất kỳ trong số các cạnh còn lại thì sẽ tạo thành chu trình đơn. Trong trường hợp này đồ thị G là không liên thông, việc tìm kiếm cây khung thất bại.

Như vậy có hai vấn đề quan trọng khi cài đặt thuật toán Kruskal:

Thứ nhất, làm thế nào để xét được các cạnh từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn. Ta có thể thực hiện bằng cách sắp xếp danh sách cạnh theo thứ tự không giảm của trọng số, sau đó duyệt từ đầu tới cuối danh sách cạnh. Trong trường hợp tổng quát, thuật toán HeapSort là hiệu quả nhất bởi nó cho phép chọn lần lượt các cạnh từ cạnh trọng nhỏ nhất tới cạnh trọng số lớn nhất ra khỏi Heap và có thể xử lý (bỏ qua hay thêm vào cây) luôn.

Thứ hai, làm thế nào kiểm tra xem việc thêm một cạnh có tạo thành chu trình đơn trong T hay không. Để ý rằng các cạnh trong T ở các bước sẽ tạo thành một rừng (đồ thị không có chu trình đơn). Muốn thêm một cạnh (u, v) vào T mà không tạo thành chu trình đơn thì (u, v) phải nối hai cây khác nhau của rừng T, bởi nếu u, v thuộc cùng một cây thì sẽ tạo thành chu trình đơn trong cây đó. Ban đầu, ta khởi tạo rừng T gồm n cây, mỗi cây chỉ gồm đúng một đỉnh, sau đó, mỗi khi xét đến **cạnh nối hai cây khác nhau** của rừng T thì ta **kết nạp cạnh đó** vào T, đồng thời **hợp nhất hai cây đó lại thành một cây**.

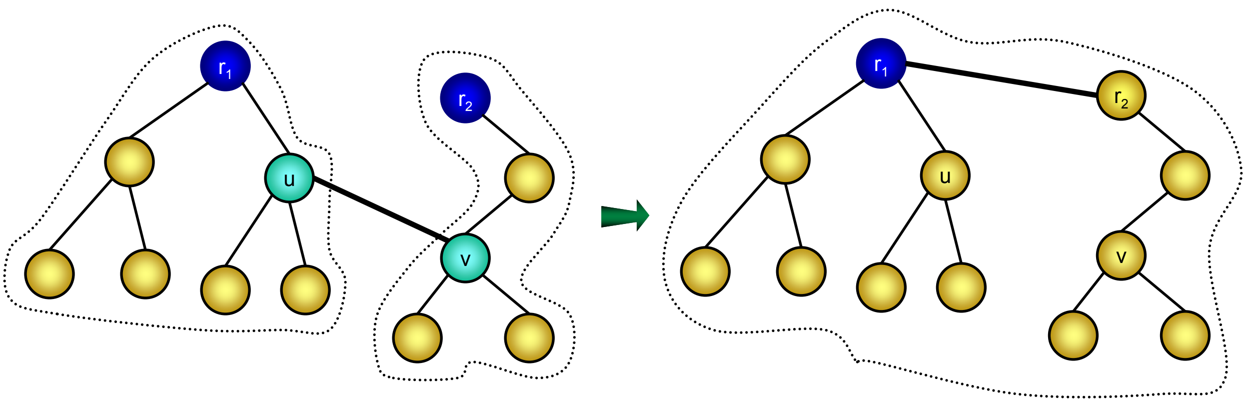
Nếu cho mỗi đỉnh v trên cây một nhãn Lab[v] là số hiệu đỉnh cha của đỉnh v trong cây, trong trường hợp v là gốc của một cây thì Lab[v] được gán một giá trị âm. Khi đó ta hoàn toàn có thể xác định được gốc của cây chứa đỉnh v bằng hàm GetRoot dưới đây:

```
function GetRoot(v∈V): ∈V;
begin
  while Lab[v] > 0 do v := Lab[v];
  GetRoot := v;
end;
```

Vậy để kiểm tra một cạnh (u, v) có nối hai cây khác nhau của rừng T hay không? ta có thể kiểm tra GetRoot(u) có khác GetRoot(v) hay không, bởi mỗi cây chỉ có duy nhất một gốc.

Để hợp nhất cây gốc r_1 và cây gốc r_2 thành một cây, ta lưu ý rằng mỗi cây ở đây chỉ dùng để ghi nhận một tập hợp đỉnh thuộc cây đó chứ cấu trúc cạnh trên cây thế nào thì không quan trọng. Vậy để hợp nhất cây gốc r_1 và cây gốc r_2, ta chỉ việc coi r_1 là nút cha của r_2 trong cây bằng cách đặt:

$$Lab[r_2] := r_1.$$



Hình 77: Hai cây gốc r_1 và r_2 và cây mới khi hợp nhất chúng

Tuy nhiên, để thuật toán làm việc hiệu quả, tránh trường hợp cây tạo thành bị suy biến khiến cho hàm GetRoot hoạt động chậm, người ta thường đánh giá: Để hợp hai cây lại thành một, thì gốc cây nào ít nút hơn sẽ bị coi là con của gốc cây kia.

Thuật toán hợp nhất cây gốc r_1 và cây gốc r_2 có thể viết như sau:

```
{Count[k] là số đỉnh của cây gốc k}
procedure Union(r1, r2 ∈ V);
begin
  if Count[r1] < Count[r2] then {Hợp nhất thành cây gốc r2}
    begin
      Count[r2] := Count[r1] + Count[r2];
      Lab[r1] := r2;
    end
  else {Hợp nhất thành cây gốc r1}
    begin
      Count[r1] := Count[r1] + Count[r2];
      Lab[r2] := r1;
    end;
end;
```

Khi cài đặt, ta có thể tận dụng ngay nhãn Lab[r] để lưu số đỉnh của cây gốc r, bởi như đã giải thích ở trên, Lab[r] chỉ cần mang một giá trị âm là đủ, vậy ta có thể coi Lab[r] = -Count[r] với r là gốc của một cây nào đó.

```
procedure Union(r1, r2 ∈ V); {Hợp nhất cây gốc r1 với cây gốc r2}
begin
  x := Lab[r1] + Lab[r2]; {-x là tổng số nút của cả hai cây}
  if Lab[r1] > Lab[r2] then {Cây gốc r1 ít nút hơn cây gốc r2, hợp nhất thành cây gốc r2}
    begin
      Lab[r1] := r2; {r2 là cha của r1}
      Lab[r2] := x; {r2 là gốc cây mới, -Lab[r2] giờ đây là số nút trong cây mới}
    end
  else {Hợp nhất thành cây gốc r1}
    begin
      Lab[r1] := x; {r1 là gốc cây mới, -Lab[r1] giờ đây là số nút trong cây mới}
      Lab[r2] := r1; {cha của r2 sẽ là r1}
    end;
end;
```

Mô hình thuật toán Kruskal:

```
for ∀k∈V do Lab[k] := -1;
for ∀(u, v)∈E (theo thứ tự từ cạnh trọng số nhỏ tới cạnh trọng số lớn) do
  begin
    r1 := GetRoot(u); r2 := GetRoot(v);
    if r1 ≠ r2 then   {(u, v) nối hai cây khác nhau}
      begin
        <Kết nạp (u, v) vào cây, nếu đã đủ n - 1 cạnh thì thuật toán dừng>
        Union(r1, r2); {Hợp nhất hai cây lại thành một cây}
      end;
  end;
```

P_4_09_1.PAS * Thuật toán Kruskal

```
program Minimal_Spanning_Tree_by_Kruskal;
const
  InputFile  = 'MINTREE.INP';
  OutputFile = 'MINTREE.OUT';
  maxV = 100;
  maxE = (maxV - 1) * maxV div 2;
type
  TEdge = record {Cấu trúc một cạnh}
    u, v, c: Integer; {Hai đỉnh và trọng số}
```

```
    Mark: Boolean; {Đánh dấu có được kết nạp vào cây khung hay không}
  end;
var
  e: array[1..maxE] of TEdge; {Danh sách cạnh}
  Lab: array[1..maxV] of Integer; {Lab[v] là đỉnh cha của v, nếu v là gốc thì Lab[v] = - số con cây gốc v}
  n, m: Integer;
  Connected: Boolean;

procedure LoadGraph;
var
  i: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n, m);
  for i := 1 to m do
    with e[i] do
      ReadLn(f, u, v, c);
  Close(f);
end;

procedure Init;
var
  i: Integer;
begin
  for i := 1 to n do Lab[i] := -1; {Rừng ban đầu, mọi đỉnh là gốc của cây gồm đúng một nút}
  for i := 1 to m do e[i].Mark := False;
end;

function GetRoot(v: Integer): Integer; {Lấy gốc của cây chứa v}
begin
  while Lab[v] > 0 do v := Lab[v];
  GetRoot := v;
end;

procedure Union(r1, r2: Integer); {Hợp nhất hai cây lại thành một cây}
var
  x: Integer;
begin
  x := Lab[r1] + Lab[r2];
  if Lab[r1] > Lab[r2] then
    begin
      Lab[r1] := r2;
      Lab[r2] := x;
    end
  else
    begin
      Lab[r1] := x;
      Lab[r2] := r1;
    end;
end;

procedure AdjustHeap(root, last: Integer); {Vun thành đống, dùng cho HeapSort}
var
  Key: TEdge;
  child: Integer;
begin
  Key := e[root];
  while root * 2 <= Last do
    begin
      child := root * 2;
      if (child < Last) and (e[child + 1].c < e[child].c)
        then Inc(child);
      if Key.c <= e[child].c then Break;
```

```
      e[root] := e[child];
      root := child;
    end;
  e[root] := Key;
end;

procedure Kruskal;
var
  i, r1, r2, Count, a: Integer;
  tmp: TEdge;
begin
  Count := 0;
  Connected := False;
  for i := m div 2 downto 1 do AdjustHeap(i, m);
  for i := m - 1 downto 0 do
    begin
      tmp := e[1]; e[1] := e[i + 1]; e[i + 1] := tmp;
      AdjustHeap(1, i);
      r1 := GetRoot(e[i + 1].u); r2 := GetRoot(e[i + 1].v);
      if r1 <> r2 then {Cạnh e[i + 1] nối hai cây khác nhau}
        begin
          e[i + 1].Mark := True; {Kết nạp cạnh đó vào cây}
          Inc(Count); {Đếm số cạnh}
          if Count = n - 1 then {Nếu đã đủ số thì thành công}
            begin
              Connected := True;
              Exit;
            end;
          Union(r1, r2); {Hợp nhất hai cây thành một cây}
        end;
    end;
end;

procedure PrintResult;
var
  i, Count, W: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  if not Connected then
    WriteLn(f, 'Error: Graph is not connected')
  else
    begin
      WriteLn(f, 'Minimal spanning tree: ');
      Count := 0;
      W := 0;
      for i := 1 to m do {Duyệt danh sách cạnh}
        with e[i] do
          begin
            if Mark then {Lọc ra những cạnh đã kết nạp vào cây khung}
              begin
                WriteLn(f, '(', u, ', ', v, ') = ', c);
                Inc(Count);
                W := W + c;
              end;
            if Count = n - 1 then Break; {Cho tới khi đủ n - 1 cạnh}
          end;
      WriteLn(f, 'Weight = ', W);
    end;
  Close(f);
end;

begin
  LoadGraph;
```

```
  Init;
  Kruskal;
  PrintResult;
end.
```

Xét về độ phức tạp tính toán, ta có thể chứng minh được rằng thao tác GetRoot có độ phức tạp là $O(\log_2 n)$, còn thao tác Union là O(1). Giả sử ta đã có danh sách cạnh đã sắp xếp rồi thì xét vòng lặp dựng cây khung, nó duyệt qua danh sách cạnh và với mỗi cạnh nó gọi 2 lần thao tác GetRoot, vậy thì độ phức tạp là $O(m\log_2 n)$, nếu đồ thị có cây khung thì $m \geq n-1$ nên ta thấy chi phí thời gian chủ yếu sẽ nằm ở thao tác sắp xếp danh sách cạnh bởi độ phức tạp của HeapSort là $O(m\log_2 m)$. Vậy độ phức tạp tính toán của thuật toán là $O(m\log_2 m)$ trong trường hợp xấu nhất. Tuy nhiên, phải lưu ý rằng để xây dựng cây khung thì ít khi thuật toán phải duyệt toàn bộ danh sách cạnh mà chỉ một phần của danh sách cạnh mà thôi.

9.3. THUẬT TOÁN PRIM (ROBERT PRIM - 1957)

Thuật toán Kruskal hoạt động chậm trong trường hợp đồ thị dày (có nhiều cạnh). Trong trường hợp đó người ta thường sử dụng phương pháp lân cận gần nhất của Prim. Thuật toán đó có thể phát biểu hình thức như sau:

Đơn đồ thị vô hướng G = (V, E) có n đỉnh được cho bởi ma trận trong số C. Qui ước $c[u, v] = +\infty$ nếu (u, v) không là cạnh. Xét cây T trong G và một đỉnh v, gọi **khoảng cách từ v tới T** là trọng số nhỏ nhất trong số các cạnh nối v với một đỉnh nào đó trong T:

$d[v] = \min\{c[u, v] \mid u \in T\}$

Ban đầu khởi tạo cây T chỉ gồm có mỗi đỉnh {1}. Sau đó cứ chọn trong số các đỉnh ngoài T ra một đỉnh gần T nhất, kết nạp đỉnh đó vào T đồng thời kết nạp luôn cả cạnh tạo ra khoảng cách gần nhất đó. Cứ làm như vậy cho tới khi:

Hoặc đã kết nạp được tất cả n đỉnh thì ta có T là cây khung nhỏ nhất

Hoặc chưa kết nạp được hết n đỉnh nhưng mọi đỉnh ngoài T đều có khoảng cách tới T là $+\infty$. Khi đó đồ thị đã cho không liên thông, ta thông báo việc tìm cây khung thất bại.

Về mặt kỹ thuật cài đặt, ta có thể làm như sau:

Sử dụng mảng đánh dấu Free. Free[v] = TRUE nếu như đỉnh v chưa bị kết nạp vào T.

Gọi d[v] là khoảng cách từ v tới T. Ban đầu khởi tạo d[1] = 0 còn $d[2] = d[3] = \ldots = d[n] = +\infty$. Tại mỗi bước chọn đỉnh đưa vào T, ta sẽ chọn đỉnh u nào ngoài T và có d[u] nhỏ nhất. Khi kết nạp u vào T rồi thì rõ ràng các nhãn d[v] sẽ thay đổi: $d[v]_{mới} := \min(d[v]_{cũ}, c[u, v])$. Vấn đề chỉ có vậy (chương trình rất giống thuật toán Dijkstra, chỉ khác ở công thức tối ưu nhãn).

P_4_09_2.PAS * Thuật toán Prim

```
program Minimal_Spanning_Tree_by_Prim;
const
  InputFile  = 'MINTREE.INP';
  OutputFile = 'MINTREE.OUT';
  max = 100;
  maxC = 10000;
```

```
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  Trace: array[1..max] of Integer; {Vết, Trace[v] là đỉnh cha của  v trong cây khung nhỏ nhất}
  n, m: Integer;
  Connected: Boolean;

procedure LoadGraph; {Nhập đồ thị}
var
  i, u, v: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, n, m);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC; {Khởi tạo ma trận trọng số}
  for i := 1 to m do
    begin
      ReadLn(f, u, v, c[u, v]);
      c[v, u] := c[u, v];
    end;
  Close(f);
end;

procedure Init;
var
  v: Integer;
begin
  d[1] := 0; {Đỉnh 1 có nhãn khoảng cách là 0}
  for v := 2 to n do d[v] := maxC; {Các đỉnh khác có nhãn khoảng cách +∞}
  FillChar(Free, SizeOf(Free), True); {Cây T ban đầu là rỗng}
end;

procedure Prim;
var
  k, i, u, v, min: Integer;
begin
  Connected := True;
  for k := 1 to n do
    begin
      u := 0; min := maxC; {Chọn đỉnh u chưa bị kết nạp có d[u] nhỏ nhất}
      for i := 1 to n do
        if Free[i] and (d[i] < min) then
          begin
            min := d[i];
            u := i;
          end;
      if u = 0 then {Nếu không chọn được u nào có d[u] < +∞  thì đồ thị không liên thông}
        begin
          Connected := False;
          Break;
        end;
      Free[u] := False; {Nếu chọn được thì đánh dấu u đã bị kết nạp, lặp lần 1 thì dĩ nhiên u = 1 bởi d[1] = 0}
      for v := 1 to n do
        if Free[v] and (d[v] > c[u, v]) then {Tính lại các nhãn khoảng cách d[v] (v chưa kết nạp)}
          begin
            d[v] := c[u, v]; {Tối ưu nhãn d[v] theo công thức}
            Trace[v] := u; {Lưu vết, đỉnh nối với v cho khoảng cách ngắn nhất là u}
          end;
    end;
end;
```

```
procedure PrintResult;
var
  v, W: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  if not Connected then {Nếu đồ thị không liên thông thì thất bại}
    WriteLn(f, 'Error: Graph is not connected')
  else
    begin
      WriteLn(f, 'Minimal spanning tree: ');
      W := 0;
      for v := 2 to n do {Cây khung nhỏ nhất gồm những cạnh (v, Trace[v])}
        begin
          WriteLn(f, '(', Trace[v], ', ', v, ') = ', c[Trace[v], v]);
          W := W + c[Trace[v], v];
        end;
      WriteLn(f, 'Weight = ', W);
    end;
  Close(f);
end;

begin
  LoadGraph;
  Init;
  Prim;
  PrintResult;
end.
```

Xét về độ phức tạp tính toán, thuật toán Prim có độ phức tạp là $O(n^2)$. Tương tự thuật toán Dijkstra, nếu kết hợp thuật toán Prim với cấu trúc Heap sẽ được một thuật toán với độ phức tạp O((m+n)logn). Tuy nhiên, nếu đồ thị có đỉnh lớn và số cạnh nhỏ, người ta thường sử dụng thuật toán Kruskal để tìm cây khung chứ không dùng thuật toán Prim với cấu trúc Heap.

Bài tập

Bài 1

So sánh hiệu quả của thuật toán Kruskal và thuật toán Prim về tốc độ.

Bài 2

Trên một nền phẳng với hệ toạ độ Decattes vuông góc đặt n máy tính, máy tính thứ i được đặt ở toạ độ (X_i, Y_i). Cho phép nối thêm các dây cáp mạng nối giữa từng cặp máy tính. Chi phí nối một dây cáp mạng tỉ lệ thuận với khoảng cách giữa hai máy cần nối. Hãy tìm cách nối thêm các dây cáp mạng để cho các máy tính trong toàn mạng là liên thông và chi phí nối mạng là nhỏ nhất.

Bài 3

Tương tự như bài 2, nhưng ban đầu đã có sẵn một số cặp máy nối rồi, cần cho biết cách nối thêm ít chi phí nhất.

Bài 4

Hệ thống điện trong thành phố được cho bởi n trạm biến thế và các đường dây điện nối giữa các cặp trạm biến thế. Mỗi đường dây điện e có độ an toàn là p(e). ở đây $0 < p(e) \le 1$. Độ an toàn của cả lưới điện là tích độ an toàn trên các đường dây. Ví dụ như có một đường dây nguy hiểm: p(e) = 1% thì cho dù các đường dây khác là tuyệt đối an toàn (độ an toàn = 100%) thì độ an toàn của mạng

cũng rất thấp (1%). Hãy tìm cách bỏ đi một số dây điện để cho các trạm biến thế vẫn liên thông và độ an toàn của mạng là lớn nhất có thể.

Bài 5

Hãy thử cài đặt thuật toán Prim với cấu trúc dữ liệu Heap chứa các đỉnh ngoài cây, tương tự như đối với thuật toán Dijkstra.

§10. BÀI TOÁN LUỒNG CỰC ĐẠI TRÊN MẠNG

Ta gọi mạng (network) là một đồ thị **có hướng** G = (V, E), trong đó có duy nhất một đỉnh A không có cung đi vào gọi là điểm phát (source), duy nhất một đỉnh B không có cung đi ra gọi là đỉnh thu (sink) và mỗi cung e = (u, v) ∈ E được gán với một số không âm c(e) = c[u, v] gọi là khả năng thông qua của cung đó (capacity). Để thuận tiện cho việc trình bày, ta qui ước rằng nếu không có cung (u, v) thì khả năng thông qua c[u, v] của nó được gán bằng 0.

Nếu có mạng G = (V, E). Ta gọi luồng (flow) f trong mạng G là một phép gán cho mỗi cung e = (u, v) ∈ E một số thực không âm f(e) = f[u, v] gọi là luồng trên cung e, thoả mãn các điều kiện sau:

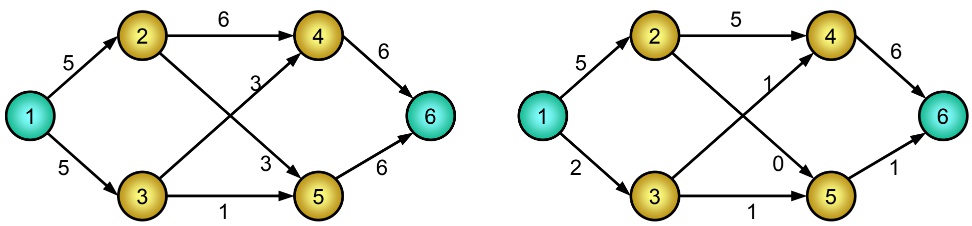
Luồng trên mỗi cung không vượt quá khả năng thông qua của nó: $0 \le f[u, v] \le c[u, v]$ ($\forall$ (u, v) ∈ E)

Với mọi đỉnh v không trùng với đỉnh phát A và đỉnh thu B, tổng luồng trên các cung đi vào v bằng tổng luồng trên các cung đi ra khỏi v: $\sum_{u\in\Gamma^-(v)} f[u,v] = \sum_{w\in\Gamma^+(v)} f[v,w]$. Trong đó:

$\Gamma^-(v) = \{u \in V \mid (u, v) \in E\}$

$\Gamma^+(v) = \{w \in V \mid (v, w) \in E\}$

Giá trị của một luồng là tổng luồng trên các cung đi ra khỏi đỉnh phát = tổng luồng trên các cung đi vào đỉnh thu.



Hình 78: Mạng với các khả năng thông qua (1 phát, 6 thu) và một luồng của nó với giá trị 7

10.1. BÀI TOÁN

Cho mạng G = (V, E). Hãy tìm luồng f* trong mạng với giá trị luồng lớn nhất. Luồng như vậy gọi là luồng cực đại trong mạng và bài toán này gọi là bài toán tìm luồng cực đại trên mạng.

10.2. LÁT CẮT, ĐƯỜNG TĂNG LUỒNG, ĐỊNH LÝ FORD - FULKERSON

10.2.1. Định nghĩa:

Ta gọi lát cắt (X, Y) là một cách phân hoạch tập đỉnh V của mạng thành hai tập rời nhau X và Y, trong đó X chứa đỉnh phát và Y chứa đỉnh thu. Khả năng thông qua của lát cắt (X, Y) là tổng tất cả các khả năng thông qua của các cung (u, v) có u ∈ X và v ∈ Y. Lát cắt với khả năng thông qua nhỏ nhất gọi là lát cắt hẹp nhất.

10.2.2. Định lý Ford-Fulkerson:

Giá trị luồng cực đại trên mạng đúng bằng khả năng thông qua của lát cắt hẹp nhất. Việc chứng minh định lý Ford- Fulkerson đã xây dựng được một thuật toán tìm luồng cực đại trên mạng:

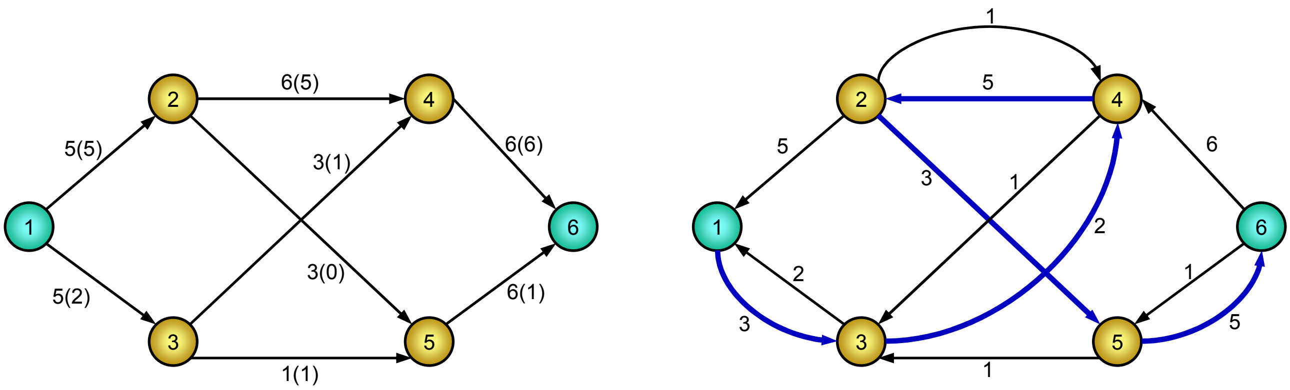
Giả sử f là một luồng trong mạng G = (V, E). Từ mạng G = (V, E) ta xây dựng đồ thị có trọng số $G_f = (V, E_f)$ như sau:

Xét những cạnh e = (u, v) ∈ E (c[u, v] > 0):

Nếu f[u, v] < c[u, v] thì ta thêm cung (u, v) vào E_f với trọng số c[u, v] - f[u, v], cung đó gọi là **cung thuận**. Về ý nghĩa, trọng số cung này cho biết còn có thể tăng luồng f trên cung (u, v) một lượng không quá trọng số đó.

Xét tiếp nếu như f[u, v] > 0 thì ta thêm cung (v, u) vào E_f với trọng số f[u, v], cung đó gọi là **cung nghịch**. Về ý nghĩa, trọng số cung này cho biết còn có thể giảm luồng f trên cung (u, v) một lượng không quá trọng số đó.

Đồ thị G_f được gọi là **đồ thị tăng luồng.**



Hình 79: Mạng G, luồng trên các cung (1 phát, 6 thu) và đồ thị tăng luồng tương ứng

Giả sử P là một đường đi cơ bản từ đỉnh phát A tới đỉnh thu B. Gọi Δ là giá trị nhỏ nhất của các trọng số của các cung trên đường đi P. Ta sẽ tăng giá trị của luồng f bằng cách đặt:

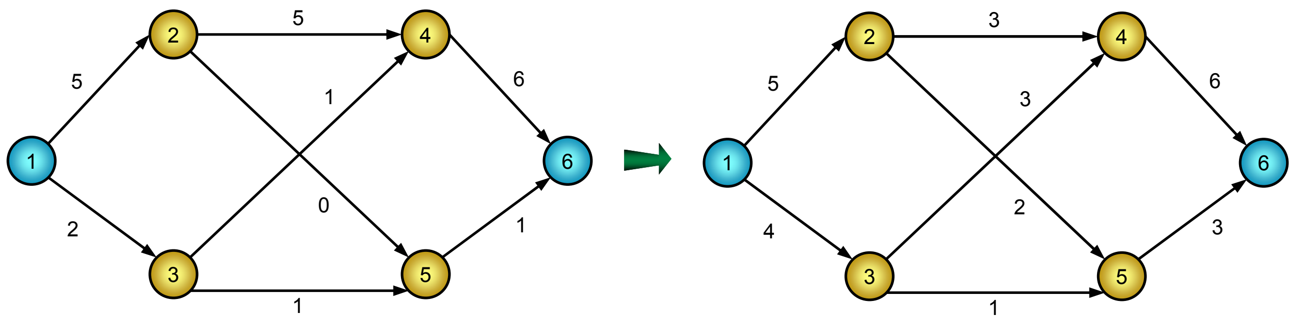
- **f[u, v] := f[u, v] + Δ**, nếu (u, v) là cung trong đường P và là cung thuận

f[v, u] := f[v, u] - Δ, nếu (u, v) là cung trong đường P và là cung nghịch

Còn luồng trên những cung khác giữ nguyên

Có thể kiểm tra luồng f mới xây dựng vẫn là luồng trong mạng và giá trị của luồng f mới được tăng thêm Δ so với giá trị luồng f cũ. Ta gọi thao tác biến đổi luồng như vậy là **tăng luồng dọc đường P**, đường đi cơ bản P từ A tới B được gọi là **đường tăng luồng.**

Ví dụ: với đồ thị tăng luồng Gf như trên, giả sử chọn đường đi (1, 3, 4, 2, 5, 6). Giá trị nhỏ nhất của trọng số trên các cung là 2, vậy thì ta sẽ tăng các giá trị f[1, 3]), f[3, 4], f[2, 5], f[5, 6] lên 2, (do các cung đó là cung thuận) và giảm giá trị f[2, 4] đi 2 (do cung (4, 2) là cung nghịch). Được luồng mới mang giá trị 9.



Hình 80: Luồng trên mạng G trước và sau khi tăng

Đến đây ta có thể hình dung ra được thuật toán tìm luồng cực đại trên mạng: khởi tạo một luồng bất kỳ, sau đó cứ **tăng luồng dọc theo đường tăng luồng, cho tới khi không tìm được đường tăng luồng nữa**

Vậy các bước của thuật toán tìm luồng cực đại trên mạng có thể mô tả như sau:

Bước 1: Khởi tạo:

Một luồng bất kỳ trên mạng, chẳng hạn như luồng 0 (luồng trên các cung đều bằng 0), sau đó:

Bước 2: Lặp hai bước sau:

Tìm đường tăng luồng P đối với luồng hiện có ≡ Tìm đường đi cơ bản từ A tới B trên đồ thị tăng luồng, nếu không tìm được đường tăng luồng thì bước lặp kết thúc.

Tăng luồng dọc theo đường P

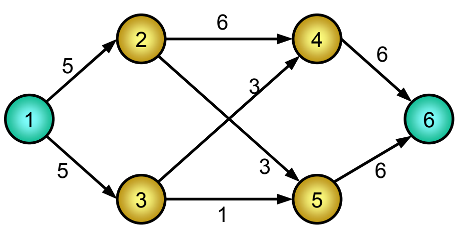
Bước 3: Thông báo giá trị luồng cực đại tìm được.

10.3. CÀI ĐẶT

Input: file văn bản MAXFLOW.INP. Trong đó:

- Dòng 1: Chứa số đỉnh n (≤ 100), số cạnh m của đồ thị, đỉnh phát A, đỉnh thu B theo đúng thứ tự cách nhau ít nhất một dấu cách
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, c[u, v] cách nhau ít nhất một dấu cách thể hiện có cung (u, v) trong mạng và khả năng thông qua của cung đó là c[u, v] (c[u, v] là số nguyên dương không quá 100)

Output: file văn bản MAXFLOW.OUT, ghi luồng trên các cung và giá trị luồng cực đại tìm được



MAXFLOW.INP	MAXFLOW.OUT
6 8 1 6	f(1, 2) = 5
1 2 5	f(1, 3) = 4
1 3 5	f(2, 4) = 3
2 4 6	f(2, 5) = 2
2 5 3	f(3, 4) = 3
3 4 3	f(3, 5) = 1
3 5 1	f(4, 6) = 6
4 6 6	f(5, 6) = 3
5 6 6	Max Flow: 9

Chú ý rằng tại mỗi bước có nhiều phương án chọn đường tăng luồng, hai cách chọn khác nhau có thể cho hai luồng cực đại khác nhau nhưng về mặt giá trị thì tất cả các luồng xây dựng được theo cách trên sẽ có cùng giá trị cực đại.

Cài đặt chương trình tìm luồng cực đại dưới đây rất chân phương, từ ma trận những khả năng thông qua c và luồng f hiện có (khởi tạo f là luồng 0), nó xây dựng đồ thị tăng luồng Gf bằng cách xây dựng ma trận cf như sau:

cf[u, v] = trọng số cung (u, v) trên đồ thị G_f nếu như (u, v) là cung thuận

cf[u, v] = - trọng số cung (u, v) trên đồ thị G_f nếu như (u, v) là cung nghịch

cf[u, v] = $+\infty$ nếu như (u, v) không phải cung của G_f

cf gần giống như ma trận trọng số của G_f, chỉ có điều ta đổi dấu trọng số nếu như gặp cung nghịch. Câu hỏi đặt ra là nếu như mạng đã cho có những đường hai chiều (có cả cung (u, v) và cung (v, u) - điều này xảy ra rất nhiều trong mạng lưới giao thông) thì đồ thị tăng luồng rất có thể là đa đồ thị (giữa u, v có thể có nhiều cung từ u tới v). Ma trận cf cũng gặp nhược điểm như ma trận trọng số: **không thể biểu diễn được đa đồ thị**, tức là nếu như có nhiều cung nối từ u tới v trong đồ thị tăng luồng thì ta đành **chấp nhận bỏ bớt mà chỉ giữ lại một cung**. Rất may cho chúng ta là điều đó không làm sai lệch đi mục đích xây dựng đồ thị tăng luồng: chỉ là tìm một đường đi từ đỉnh phát A tới đỉnh thu B mà thôi, còn đường nào thì không quan trọng.

Sau đó chương trình tìm đường đi từ đỉnh phát A tới đỉnh thu B trên đồ thị tăng luồng bằng thuật toán tìm kiếm theo chiều rộng, nếu tìm được đường đi thì sẽ tăng luồng dọc theo đường tăng luồng…

P_4_10_1.PAS * Thuật toán tìm luồng cực đại trên mạng

```
program Max_Flow;
const
  InputFile  = 'MAXFLOW.INP';
  OutputFile = 'MAXFLOW.OUT';
  max = 100;
  maxC = 10000;
var
  c, f, cf: array[1..max, 1..max] of Integer; {c: khả năng thông, f: Luồng}
  Trace: array[1..max] of Integer;
  n, A, B: Integer;

procedure Enter; {Nhập mạng}
var
  m, i, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(c, SizeOf(c), 0);
  ReadLn(fi, n, m, A, B);
  for i := 1 to m do
    ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure CreateGf; {Tìm đồ thị tăng luồng, tức là xây dựng cf từ c và f}
var
```

```
  u, v: Integer;
begin
  for u := 1 to n do
    for v := 1 to n do cf[u, v] := maxC;
  for u := 1 to n do
    for v := 1 to n do
      if c[u, v] > 0 then {Nếu u, v là cung trong mạng}
        begin
          if f[u, v] < c[u, v] then cf[u, v] := c[u, v] - f[u, v]; {Đặt cung thuận}
          if f[u, v] > 0 then cf[v, u] := -f[u, v]; {Đặt cung nghịch}
        end;
end;

{Thủ tục này tìm một đường đi từ A tới B bằng BFS, trả về TRUE nếu có đường, FALSE nếu không có đường}
function FindPath: Boolean;
var
  Queue: array[1..max] of Integer; {Hàng đợi dùng cho BFS}
  Free: array[1..max] of Boolean;
  u, v, First, Last: Integer;
begin
  FillChar(Free, SizeOf(Free), True);
  First := 1; Last := 1; Queue[1] := A; {Queue chỉ gồm một đỉnh phát A}
  Free[A] := False; {đánh dấu A}
  repeat
    u := Queue[First]; Inc(First); {Lấy u khỏi Queue}
    for v := 1 to n do
      if Free[v] and (cf[u, v] <> maxC) then {Xét v chưa đánh dấu kề với u}
        begin
          Trace[v] := u; {Lưu vết đường đi A → … → u → v}
          if v = B then    {v = B thì ta có đường đi từ A tới B, thoát thủ tục}
            begin
              FindPath := True; Exit;
            end;
          Free[v] := False; {đánh dấu v}
          Inc(Last);
          Queue[Last] := v; {Queue ← v}
        end;
  until First > Last; {Queue rỗng}
  FindPath := False; {ở trên không Exit được thì tức là không có đường}
end;

{Thủ tục tăng luồng dọc theo đường tăng luồng tìm được trong FindPath}
procedure IncFlow;
var
  u, v, IncValue: Integer;
begin
  {Trước hết dò đường theo vết để tìm trọng số nhỏ nhất của các cung trên đường}
  IncValue := maxC;
  v := B;
  while v <> A do
    begin
      u := Trace[v]; {Để ý rằng |cf[u, v]| là trọng số của cung (u, v) trên đồ thị tăng luồng}
      if Abs(cf[u, v]) < IncValue then IncValue := Abs(cf[u, v]);
      v:= u;
    end;
  {Dò lại đường lần thứ hai, lần này để tăng luồng}
  v := B;
  while v <> A do
    begin
      u := Trace[v];
      if cf[u, v] > 0 then f[u, v] := f[u, v] + IncValue {Nếu (u, v) là cung thuận trên Gf}
      else f[v, u] := f[v, u] - IncValue; {Nếu (u, v) là cung nghịch trên Gf}
      v := u;
    end;
```

```
end;

procedure PrintResult; {In luồng cực đại tìm được}
var
  u, v, m: Integer;
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  m := 0;
  for u := 1 to n do
    for v := 1 to n do
      if c[u, v] > 0 then {Nếu có cung (u, v) trên mạng thì in ra giá trị luồng f gán cho cung đó}
        begin
          WriteLn(fo, 'f(', u, ', ', v, ') = ', f[u, v]);
          if u = A then m := m + f[A, v]; {Giá trị luồng cực đại = tổng luồng phát ra từ A}
        end;
  WriteLn(fo, 'Max Flow: ', m);
  Close(fo);
end;

begin
  Enter; {Nhập dữ liệu}
  FillChar(f, SizeOf(f), 0); {Khởi tạo luồng 0}
  repeat {Bước lặp}
    CreateGf; {Dựng đồ thị tăng luồng}
    if not FindPath then Break; {Nếu không tìm được đường tăng luồng thì thoát ngay}
    IncFlow; {Tăng luồng dọc đường tăng luồng}
  until False;
  PrintResult;
end.
```

Bây giờ ta thử xem cách làm trên được ở chỗ nào và chưa hay ở chỗ nào ?

Trước hết, thuật toán tìm đường bằng Breadth First Search là khá tốt, người ta đã chứng minh rằng nếu như đường tăng luồng được tìm bằng BFS sẽ làm giảm đáng kể số bước lặp tăng luồng so với DFS.

Nhưng có thể thấy rằng việc **xây dựng tường minh cả đồ thị G_f** thông qua việc xây dựng ma trận cf chỉ để làm mỗi một việc tìm đường là lãng phí, chỉ cần dựa vào ma trận khả năng thông qua c và luồng f hiện có là ta có thể biết được (u, v) có phải là cung trên đồ thị tăng luồng G_f hay không.

Thứ hai, tại bước tăng luồng, ta phải dò lại hai lần đường đi, một lần để tìm trọng số nhỏ nhất của các cung trên đường, một lần để tăng luồng. Trong khi việc tìm trọng số nhỏ nhất của các cung trên đường có thể kết hợp làm ngay trong thủ tục tìm đường bằng cách sau:

Đặt Delta[v] là trọng số nhỏ nhất của các cung trên đường đi từ A tới v, khởi tạo Delta[A] = $+\infty$.

Tại mỗi bước từ đỉnh u thăm đỉnh v trong BFS, thì Delta[v] có thể được tính bằng giá trị nhỏ nhất trong hai giá trị Delta[u] và trọng số cung (u, v) trên đồ thị tăng luồng. Khi tìm được đường đi từ A tới B thì Delta[B] cho ta trọng số nhỏ nhất của các cung trên đường tăng luồng.

Thứ ba, ngay trong bước tìm đường tăng luồng, ta có thể xác định ngay cung nào là cung thuận, cung nào là cung nghịch. Vì vậy khi từ đỉnh u thăm đỉnh v trong BFS, ta có thể vẫn lưu vết đường đi Trace[v] := u, nhưng sau đó sẽ đổi dấu Trace[v] nếu như (u, v) là cung nghịch.

Những cải tiến đó cho ta một cách cài đặt hiệu quả hơn, đó là:

10.4. THUẬT TOÁN FORD - FULKERSON (L.R.FORD & D.R.FULKERSON - 1962)

Mỗi đỉnh v được gán nhãn (Trace[v], Delta[v]). Trong đó |Trace[v]| là đỉnh liền trước v trong đường đi từ A tới v, Trace[v] âm hay dương tuỳ theo (|Trace[v]|, v) là cung nghịch hay cung thuận trên đồ thị tăng luồng, Delta[v] là trọng số nhỏ nhất của các cung trên đường đi từ A tới v trên đồ thị tăng luồng.

Bước lặp sẽ tìm đường đi từ A tới B trên đồ thị tăng luồng đồng thời tính luôn các nhãn (Trace[v], Delta[v]). Sau đó tăng luồng dọc theo đường tăng luồng nếu tìm thấy.

P_4_10_2.PAS * Thuật toán Ford-Fulkerson

```
program Max_Flow_by_Ford_Fulkerson;
const
  InputFile  = 'MAXFLOW.INP';
  OutputFile = 'MAXFLOW.OUT';
  max = 100;
  maxC = 10000;
var
  c, f: array[1..max, 1..max] of Integer;
  Trace: array[1..max] of Integer;
  Delta: array[1..max] of Integer;
  n, A, B: Integer;

procedure Enter; {Nhập dữ liệu}
var
  m, i, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  FillChar(c, SizeOf(c), 0);
  ReadLn(fi, n, m, A, B);
  for i := 1 to m do
    ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

function Min(X, Y: Integer): Integer;
begin
  if X < Y then Min := X else Min := Y;
end;

function FindPath: Boolean;
var
  u, v: Integer;
  Queue: array[1..max] of Integer;
  First, Last: Integer;
begin
  FillChar(Trace, SizeOf(Trace), 0); {Trace[v] = 0 đồng nghĩa với v chưa đánh dấu}
  First := 1; Last := 1; Queue[1] := A;
  Trace[A] := n + 1; {Chỉ cần nó khác 0 để đánh dấu mà thôi, số dương nào cũng được cả}
  Delta[A] := maxC; {Khởi tạo nhãn}
  repeat
    u := Queue[First]; Inc(First); {Lấy u khỏi Queue}
    for v := 1 to n do
      if Trace[v] = 0 then    {Xét những đỉnh v chưa đánh dấu thăm}
        begin
          if f[u, v] < c[u, v] then    {Nếu (u, v) là cung thuận trên Gf và có trọng số là c[u, v] - f[u, v]}
```

```
            begin
              Trace[v] := u; {Lưu vết, Trace[v] mang dấu dương}
              Delta[v] := min(Delta[u], c[u, v] - f[u, v]);
            end
          else
            if f[v, u] > 0 then     {Nếu (u, v) là cung nghịch trên Gf và có trọng số là f[v, u]}
              begin
                Trace[v] := -u; {Lưu vết, Trace[v] mang dấu âm}
                Delta[v] := min(Delta[u], f[v, u]);
              end;
          if Trace[v] <> 0 then     {Trace[v] khác 0 tức là từ u có thể thăm v}
            begin
              if v = B then {Có đường tăng luồng từ A tới B}
                begin
                  FindPath := True; Exit;
                end;
              Inc(Last); Queue[Last] := v; {Đưa v vào Queue}
            end;
        end;
  until First > Last; {Hàng đợi Queue rỗng}
  FindPath := False; {ở trên không Exit được tức là không có đường}
end;

procedure IncFlow; {Tăng luồng dọc đường tăng luồng}
var
  IncValue, u, v: Integer;
begin
  IncValue := Delta[B]; {Nhãn Delta[B] chính là trọng số nhỏ nhất trên các cung của đường tăng luồng}
  v := B; {Truy vết đường đi, tăng luồng dọc theo đường đi}
  repeat
    u := Trace[v]; {Xét cung (|u|, v) trên đường tăng luồng}
    if u > 0 then f[u, v] := f[u, v] + IncValue     {(|u|, v) là cung thuận thì tăng f[u, v]}
    else
      begin
        u := -u;
        f[v, u] := f[v, u] - IncValue; {(|u|, v) là cung nghịch thì giảm f[v, |u|]}
      end;
    v := u;
  until v = A;
end;

procedure PrintResult; {In kết quả}
var
  u, v, m: Integer;
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  m := 0;
  for u := 1 to n do
    for v := 1 to n do
      if c[u, v] > 0 then
        begin
          WriteLn(fo, 'f(', u, ', ', v, ') = ', f[u, v]);
          if u = A then m := m + f[A, v];
        end;
  WriteLn(fo, 'Max Flow: ', m);
  Close(fo);
end;

begin
  Enter;
  FillChar(f, SizeOf(f), 0);
  repeat
    if not FindPath then Break;
```

```
    IncFlow;
  until False;
  PrintResult;
end.
```

Định lý về luồng cực đại trong mạng và lát cắt hẹp nhất:

Luồng cực đại trong mạng bằng khả năng thông qua của lát cắt hẹp nhất. Khi đã tìm được luồng cực đại thì theo thuật toán trên sẽ không có đường đi từ A tới B trên đồ thị tăng luồng. Nếu đặt tập X gồm những đỉnh đến được từ đỉnh phát A trên đồ thị tăng luồng (tất nhiên A∈X) và tập Y gồm những đỉnh còn lại (tất nhiên B∈Y) thì (X, Y) là lát cắt hẹp nhất đó. Có thể có nhiều lát cắt hẹp nhất, ví dụ nếu đặt tập Y gồm những đỉnh đến được đỉnh thu B trên đồ thị tăng luồng (tất nhiên B∈ Y) và tập X gồm những đỉnh còn lại thì (X, Y) cũng là một lát cắt hẹp nhất.

Định lý về tính nguyên:

Nếu tất cả các khả năng thông qua là số nguyên thì thuật toán trên luôn tìm được luồng cực đại với luồng trên cung là các số nguyên. Điều này có thể chứng minh rất dễ bởi ban đầu khởi tạo luồng 0 thì tức các luồng trên cung là nguyên. Mỗi lần tăng luồng lên một lượng bằng trọng số nhỏ nhất trên các cung của đường tăng luồng cũng là số nguyên nên cuối cùng luồng cực đại tất sẽ phải có luồng trên các cung là nguyên.

Định lý về chi phí thời gian thực hiện giải thuật:

Trong phương pháp Ford-Fulkerson, nếu dùng đường đi ngắn nhất (qua ít cạnh nhất) từ đỉnh phát tới đỉnh thu trên đồ thị tăng luồng thì cần ít hơn n.m lần chọn đường đi để tìm ra luồng cực đại.
Edmonds và Karp đã chứng minh tính chất này và đề nghị một phương pháp cải tiến: Tại mỗi bước, ta nên tìm đường tăng luồng sao cho giá trị tăng luồng được gia tăng nhiều nhất.

Nói chung đối với thuật toán Ford-Fulkerson, các đánh giá lý thuyết bị lệch rất nhiều so với thực tế, mặc dù với sự phân tích trong trường hợp xấu, chi phí thời gian thực hiện của thuật toán là khá lớn. Nhưng trên thực tế thì thuật toán này hoạt động rất nhanh và hiệu quả.

Bài tập:

Bài 1

Mạng với nhiều điểm phát và nhiều điểm thu: Cho một mạng gồm n đỉnh với p điểm phát A1, A2, …, Ap và q điểm thu B1, B2, …, Bq. Mỗi cung của mạng được gán khả năng thông qua là số nguyên. Các đỉnh phát chỉ có cung đi ra và các đỉnh thu chỉ có cung đi vào. Một luồng trên mạng này là một phép gán cho mỗi cung một số thực gọi là luồng trên cung đó không vượt quá khả năng thông qua và thoả mãn với mỗi đỉnh không phải đỉnh phát hay đỉnh thu thì tổng luồng đi vào bằng tổng luồng đi ra. Giá trị luồng bằng tổng luồng đi ra từ các đỉnh phát = tổng luồng đi vào các đỉnh thu. Hãy tìm luồng cực đại trên mạng.

Bài 2

Mạng với khả năng thông qua của các đỉnh và các cung: Cho một mạng với đỉnh phát A và đỉnh thu B. Mỗi cung (u, v) được gán khả năng thông qua c[u, v]. Mỗi đỉnh v khác với A và B được gán

khả năng thông qua d[v]. Một luồng trên mạng được định nghĩa như trước và thêm điều kiện: tổng luồng đi vào đỉnh v không được vượt quá khả năng thông qua d[v] của đỉnh đó. Hãy tìm luồng cực đại trên mạng.

Bài 3

Lát cắt hẹp nhất: Cho một đồ thị gồm n đỉnh và m cạnh và 2 đỉnh A, B. Hãy tìm cách bỏ đi một số ít nhất các cạnh để không còn đường đi từ A tới B.

Bài 4

Một lớp học có n bạn nam, n bạn nữ. Cho m món quà lưu niệm, ($n \le m$). Mỗi bạn có sở thích về một số món quà nào đó. Hãy tìm cách phân cho mỗi bạn nam tặng một món quà cho một bạn nữ thoả mãn:

Mỗi bạn nam chỉ tặng quà cho đúng một bạn nữ

Mỗi bạn nữ chỉ nhận quà của đúng một bạn nam

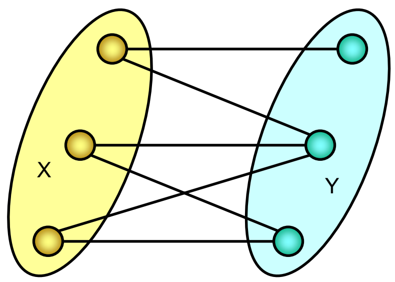
Bạn nam nào cũng đi tặng quà và bạn nữ nào cũng được nhận quà, món quà đó phải hợp sở thích của cả hai người.

Món quà nào đã được một bạn nam chọn thì bạn nam khác không được chọn nữa.

§11. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI TRÊN ĐỒ THỊ HAI PHÍA

11.1. ĐỒ THỊ HAI PHÍA (BIPARTITE GRAPH)

Các tên gọi đồ thị hai phía một dạng đơn đồ thị vô hướng G = (V, E) mà tập đỉnh của nó có thể chia làm hai tập con X, Y rời nhau sao cho bất kỳ cạnh nào của đồ thị cũng nối một đỉnh của X với một đỉnh thuộc Y. Khi đó người ta còn ký hiệu G là (X∪Y, E) và gọi một tập (chẳng hạn tập X) là **tập các đỉnh trái** và tập còn lại là **tập các đỉnh phải** của đồ thị hai phía G. Các đỉnh thuộc X còn gọi là các X_đỉnh, các đỉnh thuộc Y gọi là các Y_đỉnh.



Hình 81: Đồ thị hai phía

Để kiểm tra một đồ thị liên thông có phải là đồ thị hai phía hay không, ta có thể áp dụng thuật toán sau:

Với một đỉnh v bất kỳ:

```
X := {v}; Y := Ø;
repeat
  Y := Y ∪ Kề(X);
  X := X ∪ Kề(Y);
until  (X∩Y ≠ Ø) or (X và Y là tối đại - không bổ sung được nữa);
if X∩Y ≠ Ø then < Không phải đồ thị hai phía >
else <Đây là đồ thị hai phía, X là tập các đỉnh trái: các đỉnh đến được từ v qua một số chẵn cạnh, Y là tập các đỉnh phải: các đỉnh đến được từ v qua một số lẻ cạnh>;
```

Đồ thị hai phía gặp rất nhiều mô hình trong thực tế. Chẳng hạn quan hệ hôn nhân giữa tập những người đàn ông và tập những người đàn bà, việc sinh viên chọn trường, thầy giáo chọn tiết dạy trong thời khoá biểu v.v…

11.2. BÀI TOÁN GHÉP ĐÔI KHÔNG TRỌNG VÀ CÁC KHÁI NIỆM

Cho một đồ thị hai phía G = (X∪Y, E) ở đây X là tập các đỉnh trái và Y là tập các đỉnh phải của G

Một bộ ghép (matching) của G là một tập hợp các cạnh của G đôi một không có đỉnh chung.

Bài toán ghép đôi (matching problem) là tìm một bộ ghép lớn nhất (nghĩa là có số cạnh lớn nhất) của G

Xét một bộ ghép M của G.

Các đỉnh trong M gọi là các đỉnh đã ghép (matched vertices), các đỉnh khác là chưa ghép.

Các cạnh trong M gọi là các cạnh đã ghép, các cạnh khác là chưa ghép

Nếu định hướng lại các cạnh của đồ thị thành cung, những cạnh chưa ghép được định hướng từ X sang Y, những cạnh đã ghép định hướng từ Y về X. Trên đồ thị định hướng đó: Một đường đi xuất phát từ một X_đỉnh chưa ghép gọi là đường pha, một đường đi từ một X_đỉnh chưa ghép tới một Y_đỉnh chưa ghép gọi là đường mở.

Một cách dễ hiểu, có thể quan niệm như sau:

Một đường pha (alternating path) là một đường đi đơn trong G bắt đầu bằng một X_đỉnh chưa ghép, đi theo một cạnh **chưa ghép** sang Y, rồi đến một cạnh **đã ghép** về X, rồi lại đến một cạnh **chưa ghép** sang Y… cứ xen kẽ nhau như vậy.

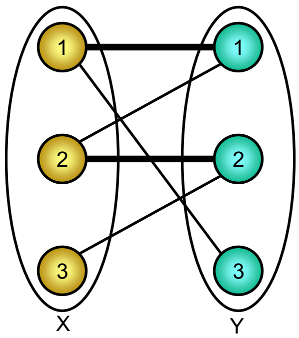
Một đường mở (augmenting path) **là một đường pha**. **Bắt đầu từ một X_đỉnh chưa ghép kết thúc bằng một Y_đỉnh chưa ghép**.

Ví dụ: với đồ thị hai phía trong hình Hình 82 và bộ ghép M = {(X1, Y1), (X2, Y2)}

X_3 và Y_3 là những đỉnh chưa ghép, các đỉnh khác là đã ghép

Đường (X_3, Y_2, X_2, Y_1) là đường pha

Đường (X_3, Y_2, X_2, Y_1, X_1, Y_3) là đường mở.



Hình 82: Đồ thị hai phía và bộ ghép M

11.3. THUẬT TOÁN ĐƯỜNG MỞ

Thuật toán đường mở để tìm một bộ ghép lớn nhất phát biểu như sau:

Bắt đầu từ một bộ ghép bất kỳ M (thông thường bộ ghép được khởi gán bằng bộ ghép rỗng hay được tìm bằng các thuật toán tham lam)

Sau đó đi tìm một đường mở, nếu tìm được thì mở rộng bộ ghép M như sau: Trên đường mở, loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép. Nếu không tìm được đường mở thì bộ ghép hiện thời là lớn nhất.

```
<Khởi tạo một bộ ghép M>;
while <Có đường mở xuất phát từ x tới một đỉnh y chưa ghép ∈Y> do
        <Dọc trên đường mở, xoá bỏ khỏi M các cạnh đã ghép và thêm vào M những cạnh chưa ghép, đỉnh x và y trở thành đã ghép, số
        cạnh đã ghép tăng lên 1>;
```

Như ví dụ trên, với bộ ghép hai cạnh M = {(X1, Y1), (X2, Y2)} và đường mở tìm được gồm các cạnh:

$(X_3, Y_2) \notin M$

$(Y_2, X_2) \in M$

$(X_2, Y_1) \notin M$

$(Y_1, X_1) \in M$

$(X_1, Y_3) \notin M$

Vậy thì ta sẽ loại đi các cạnh (Y_2, X_2) và (Y_1, X_1) trong bộ ghép cũ và thêm vào đó các cạnh (X_3, Y_2), (X_2, Y_1), (X_1, Y_3) được bộ ghép 3 cạnh.

11.4. CÀI ĐẶT

11.4.1. Biểu diễn đồ thị hai phía

Giả sử đồ thị hai phía G = (X∪Y, E) có các X_đỉnh ký hiệu là X[1], X[2], …, X[m] và các Y_đỉnh ký hiệu là Y[1], Y[2], …, Y[n]. Ta sẽ biểu diễn đồ thị hai phía này bằng ma trận A cỡ mxn. Trong đó:

A[i, j] = TRUE ⇔ có cạnh nối đỉnh X[i] với đỉnh Y[j].

11.4.2. Biểu diễn bộ ghép

Để biểu diễn bộ ghép, ta sử dụng hai mảng: matchX[1..m] và matchY[1..n].

matchX[i] là đỉnh thuộc tập Y ghép với đỉnh X[i]

matchY[j] là đỉnh thuộc tập X ghép với đỉnh Y[j].

Tức là nếu như cạnh (X[i], Y[j]) thuộc bộ ghép thì matchX[i] = j và matchY[j] = i.

Quy ước rằng:

Nếu như X[i] chưa ghép với đỉnh nào của tập Y thì matchX[i] = 0

Nếu như Y[j] chưa ghép với đỉnh nào của tập X thì matchY[j] = 0.

Để thêm một cạnh (X[i], Y[j]) vào bộ ghép thì ta chỉ việc đặt matchX[i] := j và matchY[j] := i;

Để loại một cạnh (X[i], Y[j]) khỏi bộ ghép thì ta chỉ việc đặt matchX[i] := 0 và matchY[j] := 0;

11.4.3. Tìm đường mở như thế nào.

Vì đường mở bắt đầu từ một X_đỉnh chưa ghép, đi theo một cạnh chưa ghép sang tập Y, rồi theo một đã ghép để về tập X, rồi lại một cạnh chưa ghép sang tập Y … **cuối cùng là cạnh chưa ghép** tới một Y_đỉnh chưa ghép. Nên có thể thấy ngay rằng độ dài đường mở là lẻ và trên đường mở số cạnh ∈ M ít hơn số cạnh ∉ M là 1 cạnh. Và cũng dễ thấy rằng giải thuật tìm đường mở nên sử dụng thuật toán tìm kiếm theo chiều rộng để đường mở tìm được là đường đi ngắn nhất, giảm bớt công việc cho bước tăng cặp ghép.

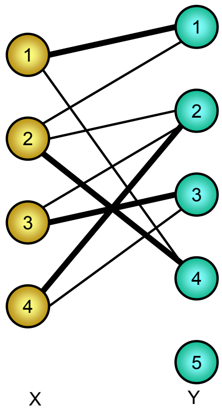
Ta khởi tạo một hàng đợi (Queue) ban đầu chứa tất cả các X_đỉnh chưa ghép. Thuật toán tìm kiếm theo chiều rộng làm việc theo nguyên tắc lấy một đỉnh v khỏi Queue và lại đẩy Queue những nối từ v chưa được thăm. Như vậy nếu thăm tới một Y_đỉnh chưa ghép thì tức là ta tìm đường mở kết thúc

ở Y_đỉnh chưa ghép đó, quá trình tìm kiếm dừng ngay. Còn nếu ta thăm tới một đỉnh j ∈ Y đã ghép, dựa vào sự kiện: **từ j chỉ có thể tới được matchY[j]** theo duy nhất một cạnh đã ghép định hướng ngược từ Y về X, nên ta có thể **đánh dấu thăm j, thăm luôn cả matchY[j], và đẩy vào Queue phần tử matchY[j] ∈ X** (Thăm liền 2 bước).

Input: file văn bản MATCH.INP

- Dòng 1: chứa hai số m, n (m, n ≤ 100) theo thứ tự là số X_đỉnh và số Y_đỉnh cách nhau ít nhất một dấu cách
- Các dòng tiếp theo, mỗi dòng ghi hai số i, j cách nhau ít nhất một dấu cách thể hiện có cạnh nối hai đỉnh (X[i], Y[j]) .

Output: file văn bản MATCH.OUT, ghi bộ ghép cực đại tìm được



MATCH.INP	MATCH.OUT
4 5	Match:
1 1	1) X[1] - Y[1]
1 4	2) X[2] - Y[4]
2 1	3) X[3] - Y[3]
2 2	4) X[4] - Y[2]
2 4	
3 2	
3 3	
4 2	
4 3	

P_4_11_1.PAS * Thuật toán đường mở tìm bộ ghép cực đại

```
program MatchingProblem;
const
  InputFile  = 'MATCH.INP';
  OutputFile = 'MATCH.OUT';
  max = 100;
var
  m, n: Integer;
  a: array[1..max, 1..max] of Boolean;
  matchX, matchY: array[1..max] of Integer;
  Trace: array[1..max] of Integer;

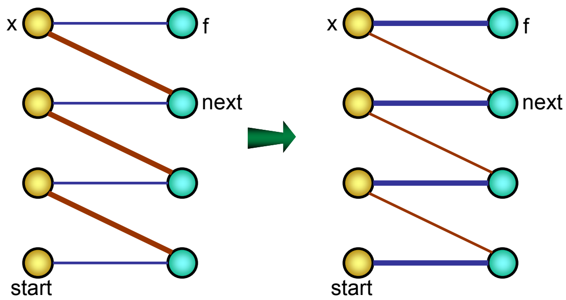
procedure Enter;
var
  i, j: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  FillChar(a, SizeOf(a), False);
  ReadLn(f, m, n);
  while not SeekEof(f) do
    begin
      ReadLn(f, i, j);
      a[i, j] := True;
    end;
  Close(f);
end;

procedure Init; {Khởi tạo bộ ghép rỗng}
begin
```

```
  FillChar(matchX, SizeOf(matchX), 0);
  FillChar(matchY, SizeOf(matchY), 0);
end;

{Tìm đường mở, nếu thấy trả về một Y_đỉnh chưa ghép là đỉnh kết thúc đường mở, nếu không thấy trả về 0}
function FindAugmentingPath: Integer;
var
  Queue: array[1..max] of Integer;
  i, j, first, last: Integer;
begin
  FillChar(Trace, SizeOf(Trace), 0); {Trace[j] = X_đỉnh liền trước Y[j] trên đường mở}
  last := 0; {Khởi tạo hàng đợi rỗng}
  for i := 1 to m do {Đẩy tất cả những X_đỉnh chưa ghép vào hàng đợi}
    if matchX[i] = 0 then
      begin
        Inc(last);
        Queue[last] := i;
      end;
  {Thuật toán tìm kiếm theo chiều rộng}
  first := 1;
  while first <= last do
    begin
      i := Queue[first]; Inc(first); {Lấy một X_đỉnh ra khỏi Queue (X[i])}
      for j := 1 to n do {Xét những Y_đỉnh chưa thăm kề với X[i] qua một cạnh chưa ghép}
        if (Trace[j] = 0) and a[i, j] and (matchX[i] <> j) then
          begin {lệnh if trên hơi thừa đk matchX[i] <> j, điều kiện Trace[j] = 0 đã bao hàm luôn điều kiện này rồi}
            Trace[j] := i; {Lưu vết đường đi}
            if matchY[j] = 0 then {Nếu j chưa ghép thì ghi nhận đường mở và thoát ngay}
              begin
                FindAugmentingPath := j;
                Exit;
              end;
            Inc(last); {Đẩy luôn matchY[j] vào hàng đợi}
            Queue[last] := matchY[j];
          end;
    end;
  FindAugmentingPath := 0; {Ở trên không Exit được tức là không còn đường mở}
end;

{Nới rộng bộ ghép bằng đường mở kết thúc ở f∈Y}
procedure Enlarge(f: Integer);
var
  x, next: Integer;
begin
  repeat
    x := Trace[f];
    next := matchX[x];
    matchX[x] := f;
    matchY[f] := x;
    f := next;
  until f = 0;
end;
```




```
procedure Solve; {Thuật toán đường mở}
var
  finish: Integer;
begin
  repeat
    finish := FindAugmentingPath; {Đầu tiên thử tìm một đường mở}
    if finish <> 0 then Enlarge(finish); {Nếu thấy thì tăng cặp và lặp lại}
  until finish = 0; {Nếu không thấy thì dừng}
end;

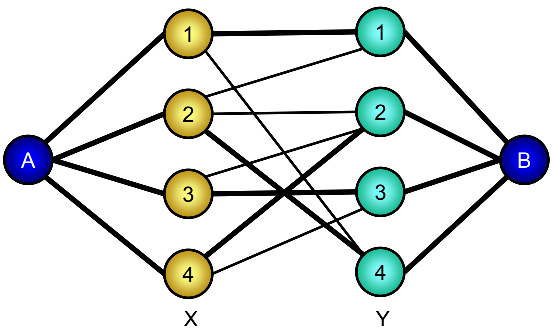
procedure PrintResult; {In kết quả}
```

```
var
  i, Count: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  WriteLn(f, 'Match: ');
  Count := 0;
  for i := 1 to m do
    if matchX[i] <> 0 then
      begin
        Inc(Count);
        WriteLn(f, Count, ') X[', i, '] - Y[', matchX[i], ']');
      end;
  Close(f);
end;

begin
  Enter;
  Init;
  Solve;
  PrintResult;
end.
```

Khảo sát tính đúng đắn của thuật toán cho ta một kết quả khá thú vị:

Nếu ta thêm một đỉnh A và cho thêm m cung từ A tới tất cả những đỉnh của tập X, thêm một đỉnh B và nối thêm n cung từ tất cả các đỉnh của Y tới B. Ta được một mạng với đỉnh phát A và đỉnh thu B.



Hình 83: Mô hình luồng của bài toán tìm bộ ghép cực đại trên đồ thị hai phía

Nếu đặt khả năng thông qua của các cung đều là 1 sau đó tìm luồng cực đại trên mạng bằng thuật toán Ford-Fulkerson thì theo định lý về tính nguyên, luồng tìm được trên các cung đều phải là số nguyên (tức là bằng 1 hoặc 0). Khi đó dễ thấy rằng những cung có luồng 1 từ tập X tới tập Y sẽ cho ta một bộ ghép lớn nhất. Để chứng minh thuật toán đường mở tìm được bộ ghép lớn nhất sau hữu hạn bước, ta sẽ chứng minh rằng số bộ ghép tìm được bằng thuật toán đường mở sẽ bằng giá trị luồng cực đại nói trên, điều đó cũng rất dễ bởi vì nếu để ý kỹ một chút thì đường mở chẳng qua là đường tăng luồng trên đồ thị tăng luồng mà thôi, ngay cái tên augmenting path đã cho ta biết điều này. Vì vậy thuật toán đường mở ở trường hợp này là một **cách cài đặt hiệu quả trên một dạng đồ thị đặc biệt**, nó làm cho chương trình sáng sủa hơn nhiều so với phương pháp tìm bộ ghép dựa trên bài toán luồng và thuật toán Ford-Fulkerson thuần túy.

Người ta đã chứng minh được chi phí thời gian thực hiện giải thuật này trong trường hợp xấu nhất sẽ là $O(n^3)$ đối với đồ thị dày và O(n(n + m)logn) đối với đồ thị thưa. Tuy nhiên, cũng giống như thuật toán Ford-Fulkerson, trên thực tế phương pháp này hoạt động rất nhanh.

Bài tập

Bài 1

Có n thợ và n công việc ($n \le 100$), mỗi thợ thực hiện được ít nhất một việc. Như vậy một thợ có thể làm được nhiều việc, và một việc có thể có nhiều thợ làm được. Hãy phân công n thợ thực hiện n việc đó sao cho mỗi thợ phải làm đúng 1 việc hoặc thông báo rằng không có cách phân công nào thoả mãn điều trên.

Bài 2

Có n thợ và m công việc ($n, m \le 100$). Mỗi thợ cho biết mình có thể làm được những việc nào, hãy phân công các thợ làm các công việc đó sao cho mỗi thợ phải làm ít nhất 2 việc và số việc thực hiện được là nhiều nhất.

Bài 3

Có n thợ và m công việc ($n, m \le 100$). Mỗi thợ cho biết mình có thể làm được những việc nào, hãy phân công thực hiện các công việc đó sao cho số công việc phân cho người thợ làm nhiều nhất thực hiện là cực tiểu.

§12. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI VỚI TRỌNG SỐ CỰC TIỂU TRÊN ĐỒ THỊ HAI PHÍA - THUẬT TOÁN HUNGARI

12.1. BÀI TOÁN PHÂN CÔNG

Đây là một dạng bài toán phát biểu như sau: Có m người (đánh số 1, 2, …, m) và n công việc (đánh số 1, 2, …, n), mỗi người có khả năng thực hiện một số công việc nào đó. Để giao cho người i thực hiện công việc j cần một chi phí là c[i, j] ≥ 0. Cần phân cho mỗi thợ một việc và mỗi việc chỉ do một thợ thực hiện sao cho số công việc có thể thực hiện được là nhiều nhất và nếu có ≥ 2 phương án đều thực hiện được nhiều công việc nhất thì chỉ ra phương án chi phí ít nhất.

Dựng đồ thị hai phía G = (X∪Y, E) với X là tập m người, Y là tập n việc và (u, v) ∈ E với trọng số c[u, v] nếu như người u làm được công việc v. Bài toán đưa về tìm bộ ghép nhiều cạnh nhất của G có trọng số nhỏ nhất.

Gọi k = max(m, n). Bổ sung vào tập X và Y một số đỉnh giả để $|X| = |Y| = k$.

Gọi M là một số dương đủ lớn hơn chi phí của mọi phép phân công có thể. Với mỗi cặp đỉnh (u, v): u ∈ X và v ∈ Y. Nếu (u, v) ∉ E thì ta bổ sung cạnh (u, v) vào E với trọng số là M.

Khi đó ta được G là một **đồ thị hai phía đầy đủ** (Đồ thị hai phía mà giữa một đỉnh bất kỳ của X và một đỉnh bất kỳ của Y đều có cạnh nối). Và nếu như ta **tìm được bộ ghép đầy đủ k cạnh mang trọng số nhỏ nhất** thì ta chỉ cần **loại bỏ khỏi bộ ghép đó những cạnh mang trọng số M vừa thêm vào** thì sẽ được kế hoạch phân công 1 người ↔ 1 việc cần tìm. Điều này dễ hiểu bởi bộ ghép đầy đủ mang trọng số nhỏ nhất tức là phải ít cạnh trọng số M nhất, tức là số phép phân công là nhiều nhất, và tất nhiên trong số các phương án ghép ít cạnh trọng số M nhất thì đây là phương án trọng số nhỏ nhất, tức là tổng chi phí trên các phép phân công là ít nhất.

12.2. PHÂN TÍCH

Vào: Đồ thị hai phía đầy đủ G = (X∪Y, E); $|X| = |Y| = k$. Được cho bởi ma trận vuông C cỡ kxk, c[i, j] = trọng số cạnh nối đỉnh X_i với Y_j. Giả thiết c[i, j] ≥ 0. với mọi i, j.

Ra: Bộ ghép đầy đủ trọng số nhỏ nhất.

Hai định lý sau đây tuy rất đơn giản nhưng là những định lý quan trọng tạo cơ sở cho thuật toán sẽ trình bày:

Định lý 1: Loại bỏ khỏi G những cạnh trọng số > 0. Nếu những cạnh trọng số 0 còn lại tạo ra bộ ghép k cạnh trong G thì đây là bộ ghép cần tìm.

Chứng minh: Theo giả thiết, các cạnh của G mang trọng số không âm nên bất kỳ bộ ghép nào trong G cũng có trọng số không âm, mà bộ ghép ở trên mang trọng số 0, nên tất nhiên đó là bộ ghép đầy đủ trọng số nhỏ nhất.

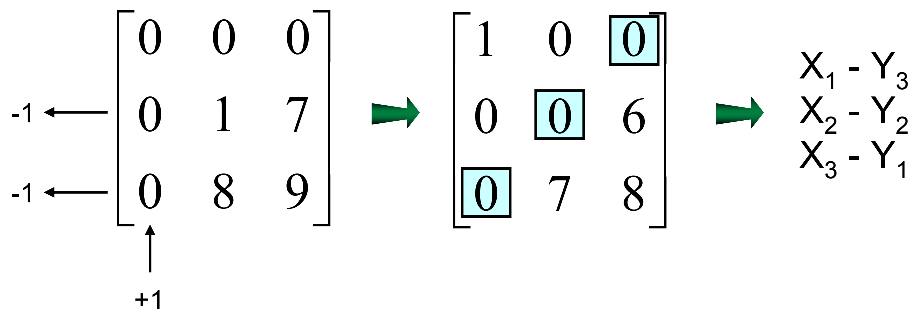
***Định lý 2:** Với đỉnh X_i, nếu ta cộng thêm một số Δ(dương hay âm) vào tất cả những cạnh liên thuộc với X_i (tương đương với việc cộng thêm Δ vào tất cả các phần tử thuộc hàng i của ma trận C) thì không ảnh hưởng tới bộ ghép đầy đủ trọng số nhỏ nhất.*

Chứng minh: Với một bộ ghép đầy đủ bất kỳ thì có một và chỉ một cạnh ghép với X[i]. Nên việc cộng thêm Δ vào tất cả các cạnh liên thuộc với X[i] sẽ làm tăng trọng số bộ ghép đó lên Δ. Vì vậy nếu như ban đầu, M là bộ ghép đầy đủ trọng số nhỏ nhất thì sau thao tác trên, M vẫn là bộ ghép đầy đủ trọng số nhỏ nhất.

Hệ quả: Với đỉnh Y[j], nếu ta cộng thêm một số Δ (dương hay âm) vào tất cả những cạnh liên thuộc với Y[j] (tương đương với việc cộng thêm Δ vào tất cả các phần tử thuộc cột j của ma trận C) thì không ảnh hưởng tới bộ ghép đầy đủ trọng số nhỏ nhất.

***Từ đây có thể nhận ra tư tưởng của thuật toán:** Từ đồ thị G, ta tìm chiến lược cộng / trừ một cách hợp lý trọng số của các cạnh liên thuộc với một đỉnh nào đó để được một đồ thị mới vẫn có các cạnh trọng số không âm, mà các cạnh trọng số 0 của đồ thị mới đó chứa một bộ ghép đầy đủ k cạnh.*

Ví dụ: Biến đổi ma trận trọng số của đồ thị hai phía 3 đỉnh trái, 3 đỉnh phải:



Hình 84: Phép xoay trọng số cạnh

12.3. THUẬT TOÁN

12.3.1. Các khái niệm:

Để cho gọn, ta gọi những cạnh trọng số 0 của G là những 0_cạnh.

Xét một bộ ghép M chỉ gồm những 0_cạnh.

Những đỉnh ∈ M gọi là những đỉnh đã ghép, những đỉnh còn lại gọi là những đỉnh chưa ghép.

Những 0_cạnh ∈ M gọi là những 0_cạnh đã ghép, những 0_cạnh còn lại là những 0_cạnh chưa ghép.

Nếu ta định hướng lại các 0_cạnh như sau: Những 0_cạnh chưa ghép cho hướng từ tập X sang tập Y, những 0_cạnh đã ghép cho hướng từ tập Y về tập X. Khi đó:

Đường pha (Alternating Path) là một đường đi cơ bản xuất phát từ một X_đỉnh chưa ghép đi theo các 0_cạnh đã định hướng ở trên. Như vậy dọc trên đường pha, các 0_cạnh chưa ghép và những

0_cạnh đã ghép xen kẽ nhau. Vì đường pha chỉ là đường đi cơ bản trên đồ thị định hướng nên việc xác định những đỉnh nào có thể đến được từ $x \in X$ bằng một đường pha có thể sử dụng các thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS). Những đỉnh và những cạnh được duyệt qua tạo thành một cây pha gốc x

Một đường mở (Augmenting Path) là một đường pha đi từ một X_đỉnh chưa ghép tới một Y_đỉnh chưa ghép. Như vậy:

Đường đi trực tiếp từ một X_đỉnh chưa ghép tới một Y_đỉnh chưa ghép qua một 0_cạnh chưa ghép cũng là một đường mở.

Dọc trên đường mở, số 0_cạnh chưa ghép nhiều hơn số 0_cạnh đã ghép đúng 1 cạnh.

12.3.2. Thuật toán Hungari

Bước 1: Khởi tạo:

Một bộ ghép M := ∅

Bước 2: Với mọi đỉnh $x^* \in X$, ta tìm cách ghép x^* như sau.

Bắt đầu từ đỉnh x^* chưa ghép, thử tìm đường mở bắt đầu ở x^* bằng thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS - thông thường nên dùng BFS để tìm đường qua ít cạnh nhất) có hai khả năng xảy ra:

Hoặc tìm được đường mở thì dọc theo đường mở, ta loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép, ta được một **bộ ghép mới nhiều hơn bộ ghép cũ 1 cạnh** và **đỉnh x^* trở thành đã ghép**.

Hoặc không tìm được đường mở thì do ta sử dụng thuật toán tìm kiếm trên đồ thị nên có thể xác định được hai tập:

VisitedX = {Tập những X_đỉnh có thể đến được từ x* bằng một đường pha}

- VisitedY = {Tập những Y_đỉnh có thể đến được từ x* bằng một đường pha}

Gọi Δ là trọng số nhỏ nhất của các cạnh nối giữa một đỉnh thuộc VisitedX với một đỉnh không thuộc VisitedY. Dễ thấy Δ > 0 bởi nếu Δ = 0 thì tồn tại một 0_cạnh (x, y) với $x \in$ VisitedX và $y \notin$ VisitedY. Vì x^* đến được x bằng một đường pha và (x, y) là một 0_cạnh nên x^* cũng đến được y bằng một đường pha, dẫn tới $y \in$ VisitedY, điều này vô lý.

Biến đổi đồ thị G như sau: Với $\forall x \in$ VisitedX, trừ Δ vào trọng số những cạnh liên thuộc với x, Với $\forall y \in$ VisitedY, cộng Δ vào trọng số những cạnh liên thuộc với y.

Lặp lại thủ tục tìm kiếm trên đồ thị thử tìm đường mở xuất phát ở x^* cho tới khi tìm ra đường mở.

Bước 3: Sau bước 2 thì mọi X_đỉnh đều được ghép, in kết quả về bộ ghép tìm được.

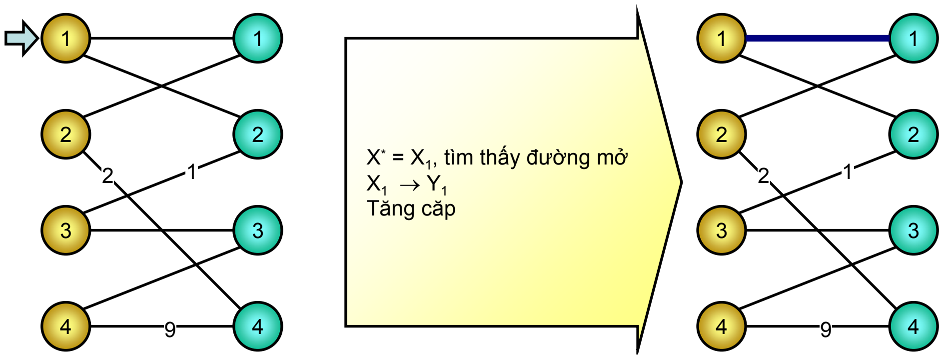
Mô hình cài đặt của thuật toán có thể viết như sau:

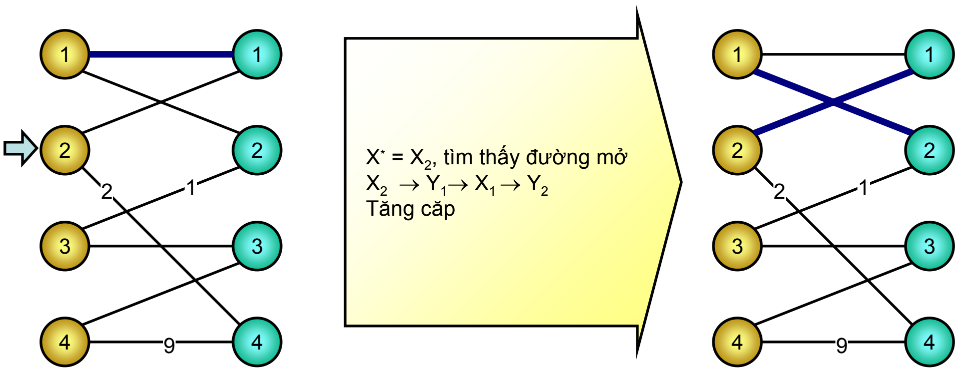
```
<Khởi tạo: M := ∅ …>;
```

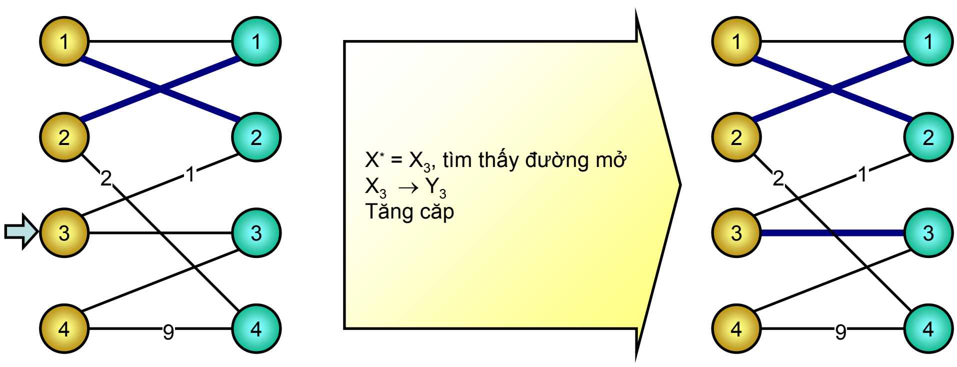
```
for (x*∈X) do
  begin
    repeat
      <Tìm đường mở xuất phát ở x*>;
      if <Không tìm thấy đường mở> then <Biến đổi đồ thị G: Chọn Δ := …>;
    until <Tìm thấy đường mở>;
    <Dọc theo đường mở, loại bỏ những cạnh đã ghép khỏi M
      và thêm vào M những cạnh chưa ghép>;
  end;
<Kết quả>;
```

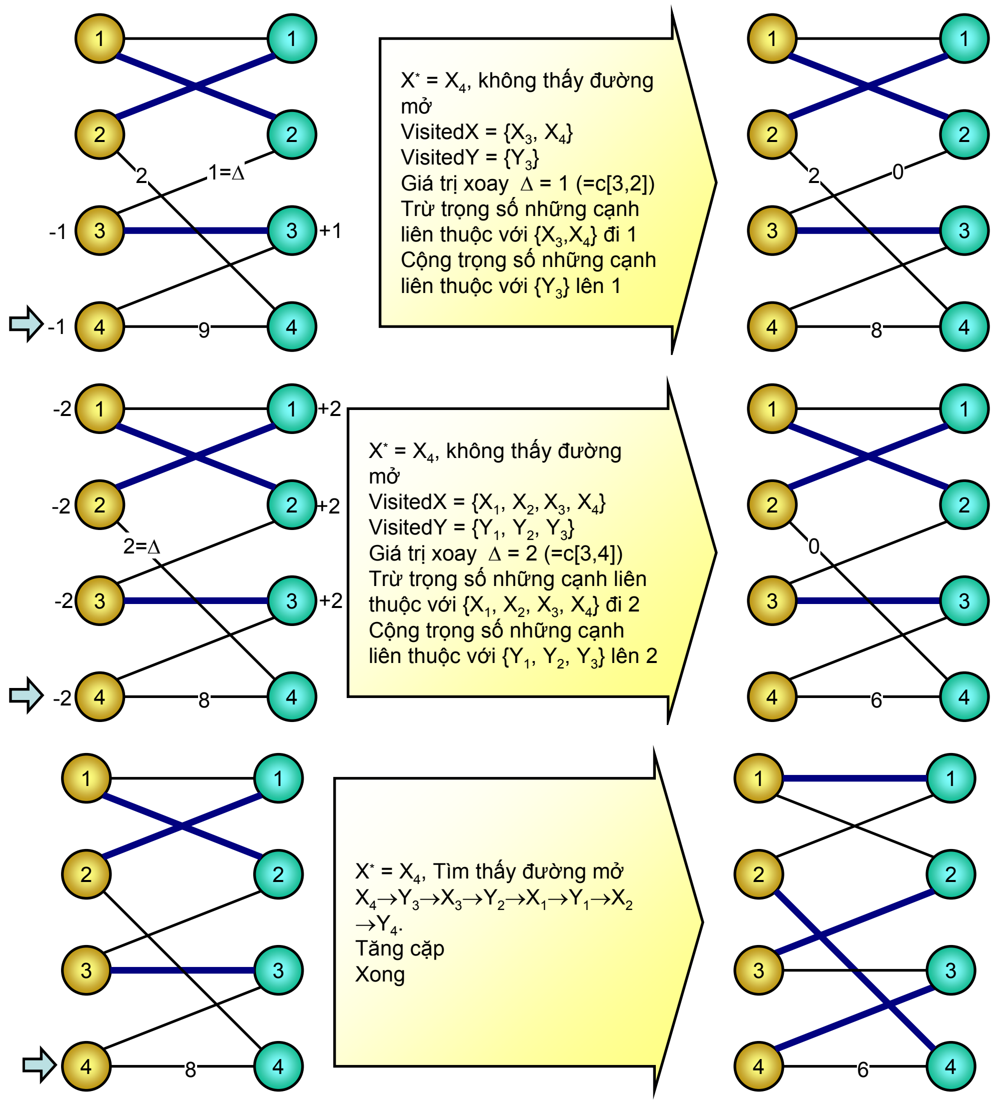
Ví dụ minh hoạ:

Để không bị rối hình, ta hiểu những cạnh không ghi trọng số là những 0_cạnh, những cạnh không vẽ mang trọng số rất lớn trong trường hợp này không cần thiết phải tính đến. Những cạnh nét đậm là những cạnh đã ghép, những cạnh nét thanh là những cạnh chưa ghép.









Hình 85: Thuật toán Hungari

Để ý rằng nếu như không tìm thấy đường mở xuất phát ở x^* thì quá trình tìm kiếm trên đồ thị sẽ cho ta một cây pha gốc x^*. Giá trị xoay Δ thực chất là trọng số nhỏ nhất của cạnh nối một X_đỉnh trong cây pha với một Y_đỉnh ngoài cây pha (cạnh ngoài). Việc trừ Δ vào những cạnh liên thuộc với X_đỉnh trong cây pha và cộng Δ vào những cạnh liên thuộc với Y_đỉnh trong cây pha sẽ làm cho cạnh ngoài nói trên trở thành 0_cạnh, các cạnh khác vẫn có trọng số ≥ 0. Nhưng quan trọng hơn là ***tất cả những cạnh trong cây pha vẫn cứ là 0_cạnh.*** *Điều đó đảm bảo cho quá trình tìm kiếm trên đồ thị lần sau sẽ xây dựng được cây pha mới lớn hơn cây pha cũ (Thể hiện ở chỗ: tập VisitedY sẽ rộng hơn trước ít nhất 1 phần tử). Vì tập các Y_ đỉnh đã ghép là hữu hạn nên sau không quá k bước, sẽ có một Y_đỉnh chưa ghép ∈ VisitedY, tức là tìm ra đường mở*

Trên thực tế, để chương trình hoạt động nhanh hơn, trong bước khởi tạo, người ta có thể thêm một thao tác:

Với mỗi đỉnh x ∈ X, xác định trọng số nhỏ nhất của các cạnh liên thuộc với x, sau đó trừ tất cả trọng số các cạnh liên thuộc với x đi trọng số nhỏ nhất đó. Làm tương tự như vậy với các Y_đỉnh.

Điều này tương đương với việc trừ tất cả các phần tử trên mỗi hàng của ma trận C đi giá trị nhỏ nhất trên hàng đó, rồi lại trừ tất cả các phần tử trên mỗi cột của ma trận C đi phần tử nhỏ nhất trên cột đó. Khi đó số 0_cạnh của đồ thị là khá nhiều, có thể chứa ngay bộ ghép đầy đủ hoặc chỉ cần qua ít bước biến đổi là sẽ chứa bộ ghép đầy đủ k cạnh.

Để tưởng nhớ hai nhà toán học König và Egervary, những người đã đặt cơ sở lý thuyết đầu tiên cho phương pháp, người ta đã lấy tên của đất nước sinh ra hai nhà toán học này để đặt tên cho thuật toán. Mặc dù sau này có một số cải tiến nhưng tên gọi Thuật toán Hungari (Hungarian Algorithm) vẫn được dùng phổ biến.

12.4. CÀI ĐẶT

12.4.1. Phương pháp đối ngẫu Kuhn-Munkres (Không làm biến đổi ma trận C ban đầu)

Phương pháp Kuhn-Munkres đi tìm hai dãy số Fx[1..k] và Fy[1..k] thoả mãn:

$c[i, j] - Fx[i] - Fy[j] \geq 0$

Tập các cạnh (X[i], Y[j]) thoả mãn c[i, j] - Fx[i] - Fy[j] = 0 chứa trọn một bộ ghép đầy đủ k cạnh, đây chính là bộ ghép cần tìm.

Chứng minh:

Nếu tìm được hai dãy số thoả mãn trên thì ta chỉ việc thực hiện hai thao tác:

Với mỗi đỉnh X[i], trừ tất cả trọng số của những cạnh liên thuộc với X[i] đi Fx[i]

Với mỗi đỉnh Y[j], trừ tất cả trọng số của những cạnh liên thuộc với Y[j] đi Fy[j]

(Hai thao tác này tương đương với việc trừ tất cả trọng số của các cạnh (X[i], Y[j]) đi một lượng Fx[i] + Fy[j] tức là c[i, j] := c[i, j] - Fx[i] - Fy[j])

Thì dễ thấy đồ thị mới tạo thành sẽ gồm có các cạnh trọng số không âm và những 0_cạnh của đồ thị chứa trọn một bộ ghép đầy đủ.

	1	2	3	4	
1	**0**	0	M	M	*Fx[1] = 2*
2	0	M	M	**2**	*Fx[2] = 2*
3	M	**1**	0	M	*Fx[3] = 3*
4	M	M	**0**	9	*Fx[4] = 3*
	Fy[1] = -2	*Fy[2] = -2*	*Fy[3] = -3*	*Fy[4] = 0*	

(Có nhiều phương án khác: Fx = (0, 0, 1, 1); Fy = (0, 0, -1, 2) cũng đúng)

Vậy phương pháp Kuhn-Munkres đưa việc biến đổi đồ thị G (biến đổi ma trận C) về việc biến đổi hay dãy số Fx và Fy. Việc trừ Δ vào trọng số tất cả những cạnh liên thuộc với X[i] tương đương với việc tăng Fx[i] lên Δ. Việc cộng Δ vào trọng số tất cả những cạnh liên thuộc với Y[j] tương đương

với giảm Fy[j] đi Δ. Khi cần biết trọng số cạnh (X[i], Y[j]) là bao nhiêu sau các bước biến đổi, thay vì viết c[i, j], ta viết c[i, j] - Fx[i] - Fy[j].

Ví dụ: Thủ tục tìm đường mở trong thuật toán Hungari đòi hỏi phải xác định được cạnh nào là 0_cạnh, khi cài đặt bằng phương pháp Kuhn-Munkres, việc xác định cạnh nào là 0_cạnh có thể kiểm tra bằng đẳng thức: c[i, j] - Fx[i] - Fy[j] = 0 hay **c[i, j] = Fx[i] + Fy[j].**

Sơ đồ cài đặt phương pháp Kuhn-Munkres có thể viết như sau:

Bước 1: Khởi tạo:

M := ∅;

Việc khởi tạo các Fx, Fy có thể có nhiều cách chẳng hạn Fx[i] := 0; Fy[j] := 0 với ∀i, j.

Hoặc: $Fx[i] := \min_{1\le j\le k}(c[i,j])$ với ∀i. Sau đó đặt $Fy[j] := \min_{1\le i\le k}(c[i,j] - Fx[i])$ với ∀j.

(Miễn sao c[i, j] - Fx[i] - Fy[j] ≥ 0)

Bước 2: Với mọi đỉnh $x^* \in X$, ta tìm cách ghép x^* như sau:

Bắt đầu từ đỉnh x^*, thử tìm đường mở bắt đầu ở x^* bằng thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS). Lưu ý rằng 0_cạnh là cạnh thoả mãn c[i, j] = Fx[i] + Fy[j]. Có hai khả năng xảy ra:

Hoặc tìm được đường mở thì dọc theo đường mở, ta loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép.

Hoặc không tìm được đường mở thì xác định được hai tập:

VisitedX = {Tập những X_đỉnh có thể đến được từ x* bằng một đường pha}

- ❖ VisitedY = {Tập những Y_đỉnh có thể đến được từ x* bằng một đường pha}
- ❖ Đặt Δ := min{c[i, j] - Fx[i] - Fy[j] | ∀X[i] ∈ VisitedX; ∀Y[j] ∉ VisitedY}

Với ∀X[i] ∈ VisitedX: Fx[i] := Fx[i] + Δ;

Với ∀Y[j] ∈ VisitedY: Fy[j] := Fy[j] - Δ;

Lặp lại thủ tục tìm đường mở xuất phát tại x^* cho tới khi tìm ra đường mở.

Đáng lưu ý ở phương pháp Kuhn-Munkres là nó không làm thay đổi ma trận C ban đầu. Điều đó thực sự hữu ích trong trường hợp trọng số của cạnh (X[i], Y[j]) không được cho một cách tường minh bằng giá trị C[i, j] mà lại cho bằng hàm c(i, j): trong trường hợp này, việc trừ hàng/cộng cột trực tiếp trên ma trận chi phí C là không thể thực hiện được.

12.4.2. Cài đặt

a) Biểu diễn bộ ghép

Để biểu diễn bộ ghép, ta sử dụng hai mảng: matchX[1..k] và matchY[1..k].

matchX[i] là đỉnh thuộc tập Y ghép với đỉnh X[i]

matchY[j] là đỉnh thuộc tập X ghép với đỉnh Y[j].

Tức là nếu như cạnh (X[i], Y[j]) thuộc bộ ghép thì matchX[i] = j và matchY[j] = i.

Quy ước rằng:

Nếu như X[i] chưa ghép với đỉnh nào của tập Y thì matchX[i] = 0

Nếu như Y[j] chưa ghép với đỉnh nào của tập X thì matchY[j] = 0.

Để thêm một cạnh (X[i], Y[j]) vào bộ ghép thì chỉ việc đặt matchX[i] := j và matchY[j] := i;

Để loại một cạnh (X[i], Y[j]) khỏi bộ ghép thì chỉ việc đặt matchX[i] := 0 và matchY[j] := 0;

b) Tìm đường mở như thế nào

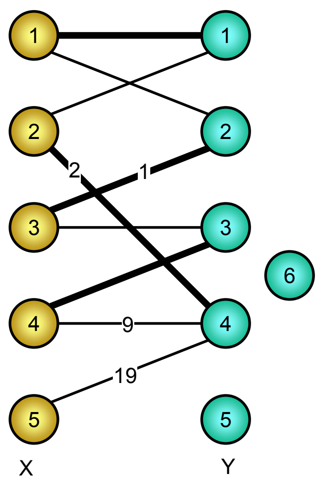
Ta sẽ tìm đường mở và xây dựng hai tập VisitedX và VisitedY bằng thuật toán tìm kiếm theo chiều rộng chỉ xét tới những đỉnh và những 0_cạnh đã định hướng như đã nói trong phần đầu:

Khởi tạo một hàng đợi (Queue) ban đầu chỉ có một đỉnh x^*. Thuật toán tìm kiếm theo chiều rộng làm việc theo nguyên tắc lấy một đỉnh v khỏi Queue và lại đẩy Queue những nối từ v chưa được thăm. Như vậy nếu thăm tới một Y_đỉnh chưa ghép thì tức là ta tìm đường mở kết thúc ở Y_đỉnh chưa ghép đó, quá trình tìm kiếm dừng ngay. Còn nếu ta thăm tới một đỉnh y ∈ Y đã ghép, dựa vào sự kiện: **từ y chỉ có thể tới được matchY[y]** theo duy nhất một 0_cạnh định hướng, nên ta có thể **đánh dấu thăm y, thăm luôn cả matchY[y], và đẩy vào Queue phần tử matchY[y] ∈ X**.

Input: file văn bản ASSIGN.INP

- Dòng 1: Ghi hai số m, n theo thứ tự là số thợ và số việc cách nhau 1 dấu cách (m, n ≤ 100)
- Các dòng tiếp theo, mỗi dòng ghi ba số i, j, c[i, j] cách nhau 1 dấu cách thể hiện thợ i làm được việc j và chi phí để làm là c[i, j] (1 ≤ i ≤ m; 1 ≤ j ≤ n; 0 ≤ c[i, j] ≤ 100).

Output: file văn bản ASSIGN.OUT, mô tả phép phân công tối ưu tìm được.



ASSIGN.INP	ASSIGN.OUT
5 6 1 1 0 1 2 0 2 1 0 2 4 2 3 2 1 3 3 0 4 3 0 4 4 9 5 4 19	Optimal assignment: 1) X[1] - Y[1] 0 2) X[2] - Y[4] 2 3) X[3] - Y[2] 1 4) X[4] - Y[3] 0 Cost: 3

P_4_12_1.PAS * Thuật toán Hungari

```
program AssignmentProblemSolve;
const
  InputFile  = 'ASSIGN.INP';
  OutputFile = 'ASSIGN.OUT';
  max = 100;
  maxC = 10001;
var
```

```
  c: array[1..max, 1..max] of Integer;
  Fx, Fy, matchX, matchY, Trace: array[1..max] of Integer;
  m, n, k, start, finish: Integer; {đường mở sẽ bắt đầu từ start∈X và kết thúc ở finish∈Y}

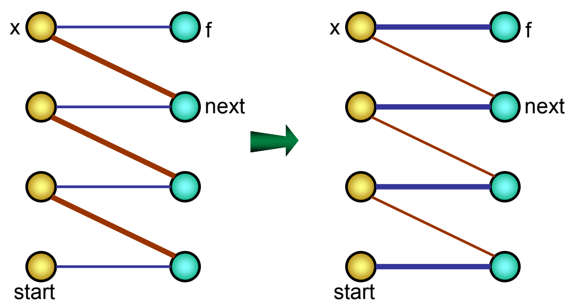
procedure Enter;
var
  i, j: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, m, n);
  if m > n then k := m else k := n;
  for i := 1 to k do
    for j := 1 to k do c[i, j] := maxC;
  while not SeekEof(f) do ReadLn(f, i, j, c[i, j]);
  Close(f);
end;

procedure Init; {Khởi tạo}
var
  i, j: Integer;
begin
  {Bộ ghép rỗng}
  FillChar(matchX, SizeOf(matchX), 0);
  FillChar(matchY, SizeOf(matchY), 0);
  {Fx[i] := Trọng số nhỏ nhất của các cạnh liên thuộc với X[i]}
  for i := 1 to k do
    begin
      Fx[i] := maxC;
      for j := 1 to k do
        if c[i, j] < Fx[i] then Fx[i] := c[i, j];
    end;
  {Fy[j] := Trọng số nhỏ nhất của các cạnh liên thuộc với Y[j]}
  for j := 1 to k do
    begin
      Fy[j] := maxC;
      for i := 1 to k do {Lưu ý là trọng số cạnh (x[i], y[j]) bây giờ là c[i, j] - Fx[i] chứ không còn là c[i, j] nữa}
        if c[i, j] - Fx[i] < Fy[j] then Fy[j] := c[i, j] - Fx[i];
    end;
  {Việc khởi tạo các Fx và Fy như thế này chỉ đơn giản là để cho số 0_cạnh trở nên càng nhiều càng tốt mà thôi}
  {Ta hoàn toàn có thể khởi gán các Fx và Fy bằng giá trị 0}
end;
{Hàm cho biết trọng số cạnh (X[i], Y[j]) }
function GetC(i, j: Integer): Integer;
begin
  GetC := c[i, j] - Fx[i] - Fy[j];
end;

procedure FindAugmentingPath; {Tìm đường mở bắt đầu ở start}
var
  Queue: array[1..max] of Integer;
  i, j, first, last: Integer;
begin
  FillChar(Trace, SizeOf(Trace), 0); {Trace[j] = X_đỉnh liền trước Y[j] trên đường mở}
  {Thuật toán BFS}
  Queue[1] := start; {Đẩy start vào hàng đợi}
  first := 1; last := 1;
  repeat
    i := Queue[first]; Inc(first); {Lấy một đỉnh X[i] khỏi hàng đợi}
    for j := 1 to k do {Duyệt những Y_đỉnh chưa thăm kề với X[i] qua một 0_cạnh chưa ghép}
      if (Trace[j] = 0) and (GetC(i, j) = 0) then
        begin
          Trace[j] := i; {Lưu vết đường đi, cùng với việc đánh dấu (≠0) luôn}
          if matchY[j] = 0 then {Nếu j chưa ghép thì ghi nhận nơi kết thúc đường mở và thoát luôn}
```

```
          begin
            finish := j;
            Exit;
          end;
        Inc(last); Queue[last] := matchY[j]; {Đẩy luôn matchY[j] vào Queue}
      end;
  until first > last; {Hàng đợi rỗng}
end;

procedure SubX_AddY; {Xoay các trọng số cạnh}
var
  i, j, t, Delta: Integer;
  VisitedX, VisitedY: set of Byte;
begin
  (*  Chú ý:
    VisitedY = {y | Trace[y] ≠ 0}
    VisitedX = {start} ∪ match(VisitedY) = {start} ∪ {matchY[y] | Trace[y] ≠ 0}
  *)
  VisitedX := [start];
  VisitedY := [];
  for j := 1 to k do
    if Trace[j] <> 0 then
      begin
        Include(VisitedX, matchY[j]);
        Include(VisitedY, j);
      end;
  {Sau khi xác định được VisitedX và VisitedY, ta tìm Δ là trọng số nhỏ nhất của cạnh nối từ VisitedX ra Y\VisitedY}
  Delta := maxC;
  for i := 1 to k do
    if i in VisitedX then
      for j := 1 to k do
        if not (j in VisitedY) and (GetC(i, j) < Delta) then
          Delta := GetC(i, j);
  {Xoay trọng số cạnh}
  for t := 1 to k do
    begin
      {Trừ trọng số những cạnh liên thuộc với VisitedX đi Delta}
      if t in VisitedX then Fx[t] := Fx[t] + Delta;
      {Cộng trọng số những cạnh liên thuộc với VisitedY lên Delta}
      if t in VisitedY then Fy[t] := Fy[t] - Delta;
    end;
end;
{Nới rộng bộ ghép bởi đường mở tìm được}
procedure Enlarge;
var
  x, next: Integer;
begin
  repeat
    x := Trace[finish];
    next := matchX[x];
    matchX[x] := finish;
    matchY[finish] := x;
    finish := Next;
  until finish = 0;
end;
```




```
procedure Solve; {Thuật toán Hungari}
var
  x, y: Integer;
begin
  for x := 1 to k do
    begin
      start := x; finish := 0; {Khởi gán nơi xuất phát đường mở, finish = 0 nghĩa là chưa tìm thấy đường mở}
```

```
      repeat
        FindAugmentingPath; {Thử tìm đường mở}
        if finish = 0 then SubX_AddY; {Nếu không thấy thì xoay các trọng số cạnh và lặp lại}
      until finish <> 0; {Cho tới khi tìm thấy đường mở}
      Enlarge; {Tăng cặp dựa trên đường mở tìm được}
    end;
end;

procedure Result;
var
  x, y, Count, W: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  WriteLn(f, 'Optimal assignment:');
  W := 0; Count := 0;
  for x := 1 to m do {In ra phép phân công thì chỉ cần xét đến m, không cần xét đến k}
    begin
      y := matchX[x];
      {Những cạnh có trọng số maxC tương ứng với một thợ không được giao việc và một việc không được phân công}
      if c[x, y] < maxC then
        begin
          Inc(Count);
          WriteLn(f, Count:5, ') X[', x, '] - Y[', y, ']  ', c[x, y]);
          W := W + c[x, y];
        end;
    end;
  WriteLn(f, 'Cost: ', W);
  Close(f);
end;

begin
  Enter;
  Init;
  Solve;
  Result;
end.
```

Nhận xét:

Nếu cài đặt như trên thì cho dù đồ thị có cạnh mang trọng số âm, chương trình vẫn tìm được bộ ghép cực đại với trọng số cực tiểu. Lý do: Ban đầu, ta trừ tất cả các phần tử trên mỗi hàng của ma trận C đi giá trị nhỏ nhất trên hàng đó, rồi lại trừ tất cả các phần tử trên mỗi cột của ma trận C đi giá trị nhỏ nhất trên cột đó (Phép trừ ở đây làm gián tiếp qua các Fx, Fy chứ không phải trừ trực tiếp trên ma trận C). Nên sau bước này, tất cả các cạnh của đồ thị sẽ có trọng số không âm bởi phần tử nhỏ nhất trên mỗi cột của C chắc chắn là 0.

Sau khi kết thúc thuật toán, tổng tất cả các phần tử ở hai dãy Fx, Fy bằng trọng số cực tiểu của bộ ghép đầy đủ tìm được trên đồ thị ban đầu.

Một vấn đề nữa phải hết sức cẩn thận trong việc ước lượng độ lớn của các phần tử Fx và Fy. Nếu như giả thiết cho các trọng số không quá 500 thì ta không thể dựa vào bất đẳng thức $Fx(x) + Fy(y) \le c(x, y)$ mà khẳng định các phần tử trong Fx và Fy cũng ≤ 500. Hãy tự tìm ví dụ để hiểu rõ hơn bản chất thuật toán.

12.5. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI VỚI TRỌNG SỐ CỰC ĐẠI TRÊN ĐỒ THỊ HAI PHÍA

Bài toán tìm bộ ghép cực đại với trọng số cực đại cũng có thể giải nhờ phương pháp Hungari bằng cách đổi dấu tất cả các phần tử ma trận chi phí (Nhờ nhận xét 1).

Khi cài đặt, ta có thể sửa lại đôi chút trong chương trình trên để giải bài toán tìm bộ ghép cực đại với trọng số cực đại mà không cần đổi dấu trọng số. Cụ thể như sau:

Bước 1: Khởi tạo:

M := ∅;

Khởi tạo hai dãy Fx và Fy thoả mãn: ∀i, j: **Fx[i] + Fy[j] ≥ c[i, j]**; Chẳng hạn ta có thể đặt Fx[i] := Phần tử lớn nhất trên dòng i của ma trận C và đặt các Fy[j] := 0.

Bước 2: Với mọi đỉnh x^*∈X, ta tìm cách ghép x^* như sau:

Với cách hiểu 0_cạnh là cạnh thoả mãn **c[i, j] = Fx[i] + Fy[j]**. Bắt đầu từ đỉnh x^*, thử tìm đường mở bắt đầu ở x^*. Có hai khả năng xảy ra:

Hoặc tìm được đường mở thì dọc theo đường mở, ta loại bỏ những cạnh đã ghép khỏi M và thêm vào M những cạnh chưa ghép.

Hoặc không tìm được đường mở thì xác định được hai tập:

VisitedX = {Tập những X_đỉnh có thể đến được từ x* bằng một đường pha}

- VisitedY = {Tập những Y_đỉnh có thể đến được từ x* bằng một đường pha}
- Đặt Δ := min{Fx[i] + Fy[j] - c[i, j] | ∀X[i] ∈ VisitedX; ∀Y[j] ∉ VisitedY}

Với ∀X[i] ∈ VisitedX: Fx[i] := Fx[i] **-** Δ;

Với ∀Y[j] ∈ VisitedY: Fy[j] := Fy[j] + Δ;

Lặp lại thủ tục tìm đường mở xuất phát tại x^* cho tới khi tìm ra đường mở.

Bước 3: Sau bước 2 thì mọi X_đỉnh đều đã ghép, ta được một bộ ghép đầy đủ k cạnh với trọng số lớn nhất.

Dễ dàng chứng minh được tính đúng đắn của phương pháp, bởi nếu ta đặt:

c'[i, j] = - c[i, j]; F'x[i] := - Fx[i]; F'y[j] = - Fy[j].

Thì bài toán trở thành tìm cặp ghép đầy đủ trọng số cực tiểu trên đồ thị hai phía với ma trận trọng số c'[1..k, 1..k]. Bài toán này được giải quyết bằng cách tính hai dãy đối ngẫu F'x và F'y. Từ đó bằng những biến đổi đại số cơ bản, ta có thể kiểm chứng được tính tương đương giữa các bước của phương pháp nêu trên với các bước của phương pháp Kuhn-Munkres ở mục trước.

12.6. NÂNG CẤP

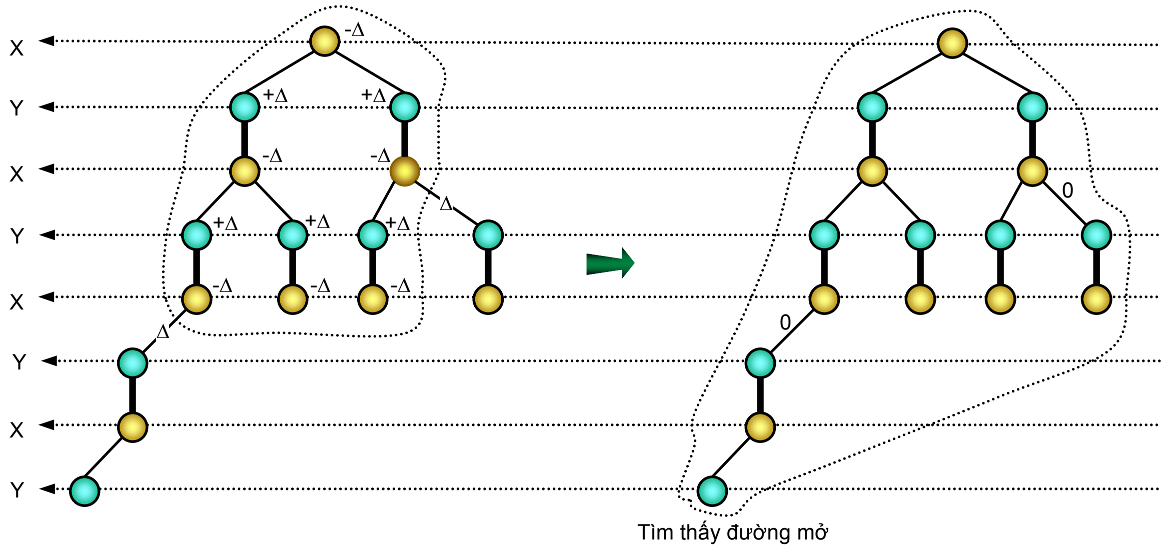
Dựa vào mô hình cài đặt thuật toán Kuhn-Munkres ở trên, ta có thể đánh giá về độ phức tạp tính toán lý thuyết của cách cài đặt này:

Thuật toán tìm kiếm theo chiều rộng được sử dụng để tìm đường mở có độ phức tạp $O(k^2)$, mỗi lần xoay trọng số cạnh mất một chi phí thời gian cỡ $O(k^2)$. Vậy mỗi lần tăng cặp, cần tối đa k lần dò đường và k lần xoay trọng số cạnh, mất một chi phí thời gian cỡ $O(k^3)$. Thuật toán cần k lần tăng cặp nên độ phức tạp tính toán trên lý thuyết của phương pháp này cỡ $O(k^4)$.

Có thể cải tiến mô hình cài đặt để được một thuật toán với độ phức tạp $O(k^3)$ dựa trên những nhận xét sau:

12.6.1. Nhận xét 1

Quá trình tìm kiếm theo chiều rộng bắt đầu từ một đỉnh x^* chưa ghép cho ta một cây pha gốc x^*. Nếu tìm được đường mở thì dừng lại và tăng cặp ngay, nếu không thì xoay trọng số cạnh và bắt đầu tìm kiếm lại để được một cây pha mới lớn hơn cây pha cũ (Hình 86):



Hình 86: Cây pha "mọc" lớn hơn sau mỗi lần xoay trọng số cạnh và tìm đường

Nhận xét 2

Việc xác định trọng số nhỏ nhất của cạnh nối một X_đỉnh trong cây pha với một Y_đỉnh ngoài cây pha có thể kết hợp ngay trong bước dựng cây pha mà không làm tăng cấp phức tạp tính toán. Để thực hiện điều này, ta sử dụng kỹ thuật như trong thuật toán Prim:

Với mọi $y \in Y$, gọi d[y] := khoảng cách từ y đến cây pha gốc x^*. Ban đầu d[y] được khởi tạo bằng trọng số cạnh $(x^*, y) = c[x^*, y] - Fx[x^*] - Fy[y]$ (cây pha ban đầu chỉ có đúng một đỉnh x^*).

Trong bước tìm đường bằng BFS, mỗi lần rút một đỉnh x ra khỏi Queue, ta xét những đỉnh $y \in Y$ chưa thăm và đặt lại $d[y]_{mới}$:= min($d[y]_{cũ}$, trọng số cạnh (x, y)) sau đó mới kiểm tra xem (x, y) có phải là 0_cạnh hay không để tiếp tục các thao tác như trước. Nếu quá trình BFS không tìm ra đường mở thì giá trị xoay Δ chính là giá trị nhỏ nhất trong các d[y] dương. Ta bớt được một đoạn chương trình tìm giá trị xoay có độ phức tạp $O(k^2)$. Công việc tại mỗi bước xoay chỉ là tìm giá trị nhỏ nhất trong các d[y] dương và thực hiện phép cộng, trừ trên hai dãy đối ngẫu Fx và Fy, nó có độ phức tạp

tính toán O(k), tối đa có k lần xoay để tìm đường mở nên tổng chi phí thời gian thực hiện các lần xoay cho tới khi tìm ra đường mở cỡ $O(k^2)$. Lưu ý rằng đồ thị đang xét là đồ thị hai phía đầy đủ nên sau khi xoay các trọng số cạnh bằng giá trị xoay Δ, tất cả các cạnh nối từ X_đỉnh trong cây pha tới Y_đỉnh ngoài cây pha đều bị giảm trọng số đi Δ, chính vì vậy ta phải trừ tất cả các d[y] > 0 đi Δ để giữ được tính hợp lý của các d[y].

Nhận xét 3:

Ta có thể tận dụng kết quả của quá trình tìm kiếm theo chiều rộng ở bước trước để nới rộng cây pha cho bước sau (grow alternating tree) mà không phải tìm lại từ đầu (BFS lại bắt đầu từ x^*).

Khi không tìm thấy đường mở, quá trình tìm kiếm theo chiều rộng sẽ đánh dấu được những đỉnh đã thăm (thuộc cây pha) và hàng đợi các X_đỉnh trong quá trình tìm kiếm trở thành rỗng. Tiếp theo là phải xác định được Δ = trọng số nhỏ nhất của cạnh nối một X_đỉnh đã thăm với một Y_đỉnh chưa thăm và xoay các trọng số cạnh để những cạnh này trở thành 0_cạnh. Tại đây ta sẽ dùng kỹ thuật sau: Thăm luôn những đỉnh $y \in Y$ chưa thăm tạo với một X_đỉnh đã thăm một 0_cạnh (những Y_đỉnh chưa thăm có d[y] = 0), nếu tìm thấy đường mở thì dừng ngay, nếu không thấy thì đẩy tiếp những đỉnh matchY[y] vào hàng đợi và lặp lại thuật toán tìm kiếm theo chiều rộng bắt đầu từ những đỉnh này. Vậy nếu xét tổng thể, mỗi lần tăng cặp ta chỉ thực hiện một lần dựng cây pha, tức là tổng chi phí thời gian của những lần thực hiện giải thuật tìm kiếm trên đồ thị sau mỗi lần tăng cặp chỉ còn là $O(k^2)$.

Nhận xét 4:

Thủ tục tăng cặp dựa trên đường mở (Enlarge) có độ phức tạp O(k)

Từ 3 nhận xét trên, phương pháp đối ngẫu Kuhn-Munkres có thể cài đặt bằng một chương trình có độ phức tạp tính toán $O(k^3)$ bởi nó cần k lần tăng cặp và chi phí cho mỗi lần là $O(k^2)$.

P_4_12_2.PAS * Cài đặt phương pháp Kuhn-Munkres $O(n^3)$

```
program AssignmentProblemSolve;
const
  InputFile  = 'ASSIGN.INP';
  OutputFile = 'ASSIGN.OUT';
  max = 100;
  maxC = 10001;
var
  c: array[1..max, 1..max] of Integer;
  Fx, Fy, matchX, matchY: array[1..max] of Integer;
  Trace, Queue, d, arg: array[1..max] of Integer;
  first, last: Integer;
  start, finish: Integer;
  m, n, k: Integer;

procedure Enter; {Nhập dữ liệu}
var
  i, j: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  ReadLn(f, m, n);
  if m > n then k := m else k := n;
```

```
  for i := 1 to k do
    for j := 1 to k do c[i, j] := maxC;
  while not SeekEof(f) do ReadLn(f, i, j, c[i, j]);
  Close(f);
end;

procedure Init; {Khởi tạo bộ ghép rỗng và hai dãy đối ngẫu Fx, Fy}
var
  i, j: Integer;
begin
  FillChar(matchX, SizeOf(matchX), 0);
  FillChar(matchY, SizeOf(matchY), 0);
  for i := 1 to k do
    begin
      Fx[i] := maxC;
      for j := 1 to k do
        if c[i, j] < Fx[i] then Fx[i] := c[i, j];
    end;
  for j := 1 to k do
    begin
      Fy[j] := maxC;
      for i := 1 to k do
        if c[i, j] - Fx[i] < Fy[j] then Fy[j] := c[i, j] - Fx[i];
    end;
end;

function GetC(i, j: Integer): Integer; {Hàm trả về trọng số cạnh (X[i], Y[j])}
begin
  GetC := c[i, j] - Fx[i] - Fy[j];
end;

procedure InitBFS; {Thủ tục khởi tạo trước khi tìm cách ghép start∈X}
var
  y: Integer;
begin
  {Hàng đợi chỉ gồm mỗi một đỉnh Start ⇔ cây pha khởi tạo chỉ có 1 đỉnh start}
  first := 1; last := 1;
  Queue[1] := start;
  {Khởi tạo các Y_đỉnh đều chưa thăm ⇔ Trace[y] = 0, ∀y}
  FillChar(Trace, SizeOf(Trace), 0);
  {Khởi tạo các d[y]}
  for y := 1 to k do
    begin
      d[y] := GetC(start, y); {d[y] là khoảng cách từ y tới cây pha gốc start}
      arg[y] := start; {arg[y] là X_đỉnh thuộc cây pha tạo ra khoảng cách đó}
    end;
  finish := 0;
end;

procedure Push(v: Integer); {Đẩy một đỉnh v∈X vào hàng đợi}
begin
  Inc(last); Queue[last] := v;
end;

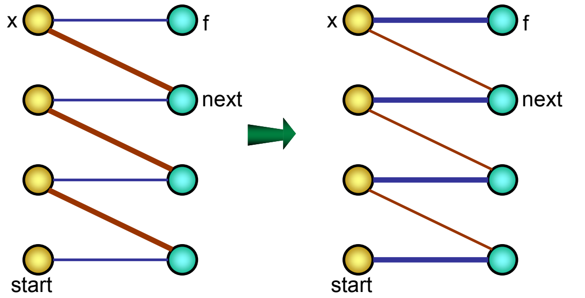
function Pop: Integer; {Rút một X_đỉnh khỏi hàng đợi, trả về trong kết quả hàm}
begin
  Pop := Queue[first]; Inc(first);
end;

procedure FindAugmentingPath; {Thủ tục tìm đường mở}
var
  i, j, w: Integer;
begin
```

```
  repeat
    i := Pop; {Rút một đỉnh X[i] khỏi hàng đợi}
    for j := 1 to k do {Quét những Y_đỉnh chưa thăm}
      if Trace[j] = 0 then
        begin
          w := GetC(i, j); {xét cạnh (X[i], Y[j])}
          if w = 0 then {Nếu là 0_cạnh}
            begin
              Trace[j] := i; {Lưu vết đường đi}
              if matchY[j] = 0 then {Nếu j chưa ghép thì ghi nhận nơi kết thúc đường mở và thoát}
                begin
                  finish := j;
                  Exit;
                end;
              Push(matchY[j]); {Nếu j đã ghép thì đẩy tiếp matchY[j] vào hàng đợi}
            end;
          if d[j] > w then {Cập nhật lại khoảng cách d[j] nếu thấy cạnh (X[i], Y[j]) ngắn hơn khoảng cách này}
            begin
              d[j] := w;
              arg[j] := i;
            end;
        end;
  until first > last;
end;

{Xoay các trọng số cạnh}
procedure SubX_AddY;
var
  Delta: Integer;
  x, y: Integer;
begin
  {Trước hết tính Δ = giá trị nhỏ nhất trọng số các d[y], với y∈Y chưa thăm (y không thuộc cây pha)}
  Delta := maxC;
  for y := 1 to k do
    if (Trace[y] = 0) and (d[y] < Delta) then Delta := d[y];
  {Trừ trọng số những cạnh liên thuộc với start∈X đi Δ}
  Fx[start] := Fx[start] + Delta;
  for y := 1 to k do {Xét các đỉnh y∈Y}
    if Trace[y] <> 0 then {Nếu y thuộc cây pha}
      begin
        x := matchY[y]; {Thì x = matchY[y] cũng phải thuộc cây pha}
        Fy[y] := Fy[y] - Delta; {Cộng trọng số những cạnh liên thuộc với y lên Δ}
        Fx[x] := Fx[x] + Delta; {Trừ trọng số những cạnh liên thuộc với x đi Δ}
      end
    else
      d[y] := d[y] - Delta; {Nếu y ∉ cây pha thì sau bước xoay, khoảng cách từ y đến cây pha sẽ giảm Δ}
  {Chuẩn bị tiếp tụcBFS}
  for y := 1 to k do
    if (Trace[y] = 0) and (d[y] = 0) then {Thăm luôn những đỉnh y∈Y tạo với cây pha một 0_cạnh}
      begin
        Trace[y] := arg[y]; {Lưu vết đường đi}
        if matchY[y] = 0 then {Nếu y chưa ghép thì ghi nhận đỉnh kết thúc đường mở và thoát ngay}
          begin
            finish := y;
            Exit;
          end;
        Push(matchY[y]); {Nếu y đã ghép thì đẩy luôn matchY[y] vào hàng đợi để chờ loang tiếp}
      end;
end;

procedure Enlarge; {Nới rộng bộ ghép bằng đường mở kết thúc ở finish}
var
  x, next: Integer;
```




```
begin
  repeat
    x := Trace[finish];
    next := matchX[x];
    matchX[x] := finish;
    matchY[finish] := x;
    finish := Next;
  until finish = 0;
end;

procedure Solve;
var
  x, y: Integer;
begin
  for x := 1 to k do {Với mỗi X_đỉnh: }
    begin
      start := x; {Đặt nơi khởi đầu đường mở}
      InitBFS; {Khởi tạo cây pha}
      repeat
        FindAugmentingPath; {Tìm đường mở}
        if finish = 0 then SubX_AddY; {Nếu không thấy thì xoay các trọng số cạnh …}
      until finish <> 0; {Cho tới khi tìm ra đường mở}
      Enlarge; {Nới rộng bộ ghép bởi đường mở tìm được}
    end;
end;

procedure Result;
var
  x, y, Count, W: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  WriteLn(f, 'Optimal assignment:');
  W := 0; Count := 0;
  for x := 1 to m do {Với mỗi X_đỉnh, xét cặp ghép tương ứng}
    begin
      y := matchX[x];
      if c[x, y] < maxC then {Chỉ quan tâm đến những cặp ghép có trọng số < maxC}
        begin
          Inc(Count);
          WriteLn(f, Count:5, ') X[', x, '] - Y[', y, ']  ', c[x, y]);
          W := W + c[x, y];
        end;
    end;
  WriteLn(f, 'Cost: ', W);
  Close(f);
end;

begin
  Enter;
  Init;
  Solve;
  Result;
end.
```

§13. BÀI TOÁN TÌM BỘ GHÉP CỰC ĐẠI TRÊN ĐỒ THỊ

13.1. CÁC KHÁI NIỆM

Xét đồ thị G = (V, E), một bộ ghép trên đồ thị G là một tập các cạnh đôi một không có đỉnh chung.

Bài toán tìm bộ ghép cực đại trên đồ thị tổng quát phát biểu như sau:

Cho một đồ thị G, phải tìm một bộ ghép cực đại trên G (bộ ghép có nhiều cạnh nhất).

Với một bộ ghép M của đồ thị G, ta gọi:

Những cạnh thuộc M được gọi là cạnh đã ghép hay **cạnh đậm**

Những cạnh không thuộc M được gọi là cạnh chưa ghép hay **cạnh nhạt**

Những đỉnh đầu mút của các cạnh đậm được gọi là **đỉnh đã ghép**, những đỉnh còn lại gọi là **đỉnh chưa ghép**

Một đường đi cơ bản (đường đi không có đỉnh lặp lại) được gọi là **đường pha** nếu nó bắt đầu bằng một cạnh nhạt và tiếp theo là các cạnh đậm, nhạt nằm nối tiếp xen kẽ nhau.

Một chu trình cơ bản (chu trình không có đỉnh trong lặp lại) được gọi là một **Blossom** nếu nó đi qua ít nhất 3 đỉnh, bắt đầu và kết thúc bằng cạnh nhạt và dọc trên chu trình, các cạnh đậm, nhạt nằm nối tiếp xen kẽ nhau. Đỉnh xuất phát của chu trình (cũng là đỉnh kết thúc) được gọi là **đỉnh cơ sở** (base) của Blossom.

Đường mở là một đường pha bắt đầu ở một đỉnh chưa ghép và kết thúc ở một đỉnh chưa ghép.

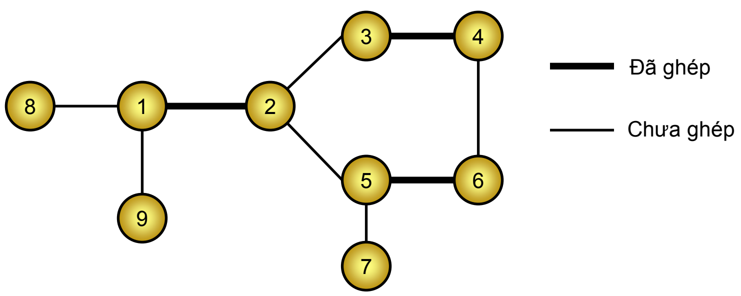
Ví dụ: Với đồ thị G và bộ ghép M trong Hình 87:

Đường (8, 1, 2, 5, 6, 4) là một đường pha

Chu trình (2, 3, 4, 6, 5, 2) là một Blossom

Đường (8, 1, 2, 3, 4, 6, 5, 7) là một đường mở

Đường (8, 1, 2, 3, 4, 6, 5, 2, 1, 9) tuy có các cạnh đậm/nhạt xen kẽ nhưng không phải đường pha (và tất nhiên không phải đường mở) vì đây không phải là đường đi cơ bản.



Hình 87: Đồ thị G và một bộ ghép M

Ta dễ dàng suy ra được các tính chất sau

Đường mở cũng như Blossom đều là đường đi độ dài lẻ với số cạnh nhạt nhiều hơn số cạnh đậm đúng 1 cạnh.

Trong mỗi Blossom, những đỉnh không phải đỉnh cơ sở đều là đỉnh đã ghép và đỉnh ghép với đỉnh đó cũng phải thuộc Blossom.

Vì Blossom là một chu trình nên trong mỗi Blossom, những đỉnh không phải đỉnh cơ sở đều tồn tại hai đường pha từ đỉnh cơ sở đi đến nó, một đường kết thúc bằng cạnh đậm và một đường kết thúc bằng cạnh nhạt, hai đường pha này được hình thành bằng cách đi dọc theo chu trình theo hai hướng ngược nhau. Như ví dụ ở Hình 87, đỉnh 4 có hai đường pha đi đỉnh cơ sở 2 đi tới: (2, 3, 4) là đường pha kết thúc bằng cạnh đậm và (2, 5, 6, 4) là đường pha kết thúc bằng cạnh nhạt

13.2. THUẬT TOÁN EDMONDS (1965)

Cơ sở của thuật toán là định lý (C.Berge): Một bộ ghép M của đồ thị G là cực đại khi và chỉ khi không tồn tại đường mở đối với M.

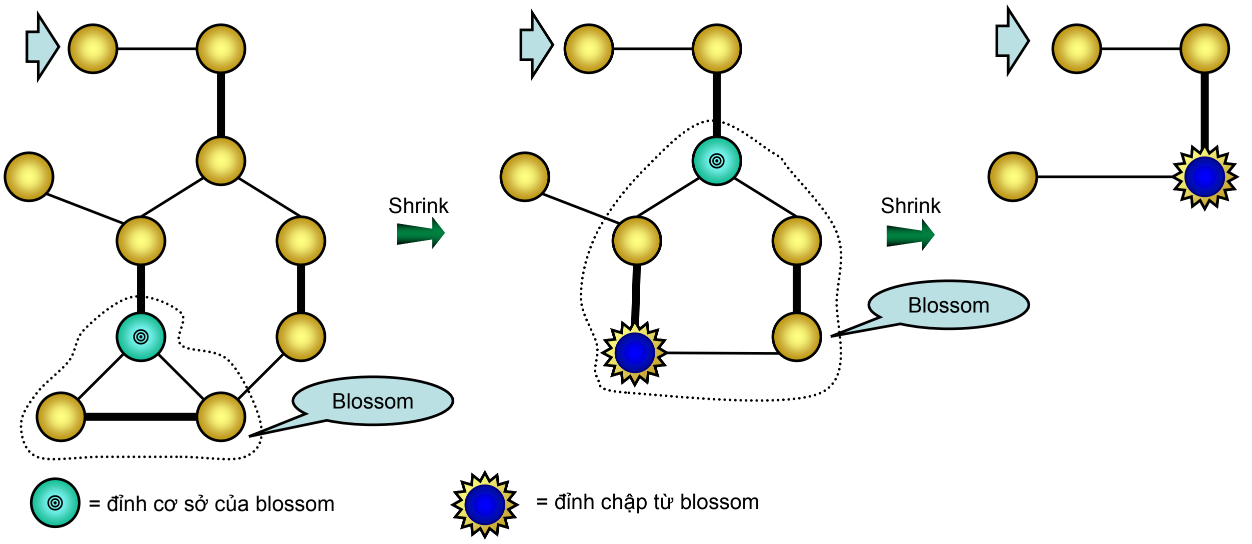
Thuật toán Edmonds:

```
M := ∅;
for (∀ đỉnh u chưa ghép) do
  if <Tìm đường mở xuất phát từ u> then
    <
     Dọc trên đường mở:
       Loại bỏ những cạnh đậm khỏi M;
       Thêm vào M những cạnh nhạt;
    >
Result: M là bộ ghép cực đại trên G
```

Điều khó nhất trong thuật toán Edmonds là phải xây dựng thuật toán tìm đường mở xuất phát từ một đỉnh chưa ghép. Thuật toán đó được xây dựng bằng cách kết hợp một thuật toán tìm kiếm trên đồ thị với phép chập Blossom.

Xét những đường pha xuất phát từ một đỉnh x chưa ghép. Những đỉnh có thể đến được từ x bằng một đường pha kết thúc là cạnh nhạt được gán nhãn "nhạt", những đỉnh có thể đến được từ x bằng một đường pha kết thúc là cạnh đậm được gán nhãn "đậm".

Với một Blossom, ta định nghĩa phép chập (shrink) là phép thay thế các đỉnh trong Blossom bằng một đỉnh duy nhất. Những cạnh nối giữa một đỉnh thuộc Blossom tới một đỉnh v nào đó không thuộc Blossom được thay thế bằng cạnh nối giữa đỉnh chập này với v và giữ nguyên tính đậm/nhạt. Dễ thấy rằng sau mỗi phép chập, các cạnh đậm vẫn được đảm bảo là bộ ghép trên đồ thị mới:



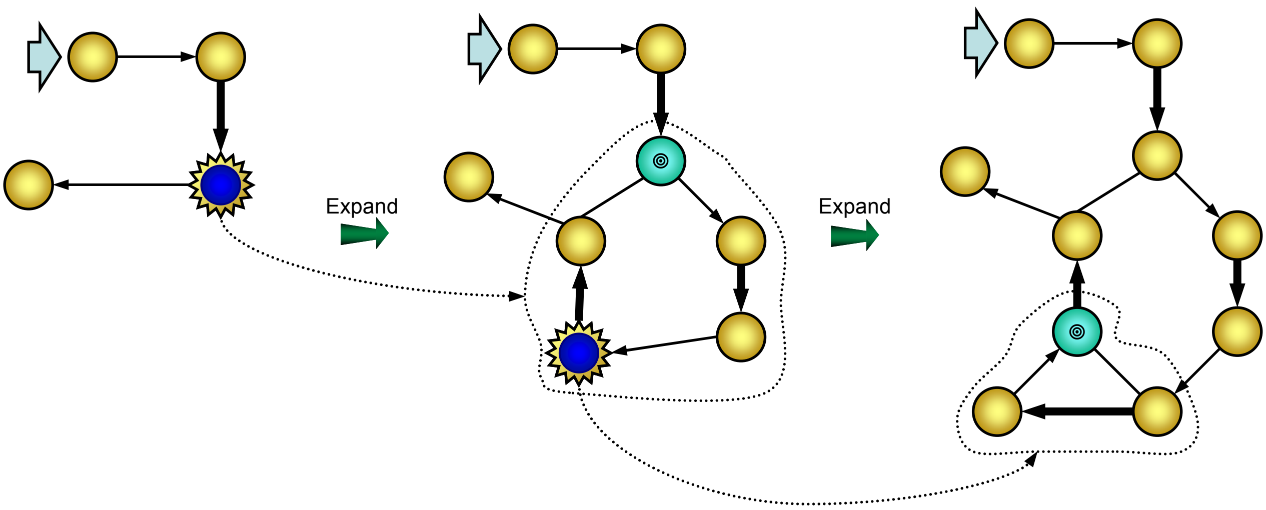
Hình 88: Phép chập Blossom

Thuật toán tìm đường mở có thể phát biểu như sau.

Trước hết đỉnh xuất phát x được gán nhãn đậm.

Tiếp theo là thuật toán tìm kiếm trên đồ thị bắt đầu từ x, theo nguyên tắc: từ đỉnh đậm chỉ được phép đi tiếp theo cạnh nhạt và từ đỉnh nhạt chỉ được đi tiếp theo cạnh đậm. Mỗi khi thăm tới một đỉnh, ta gán nhãn đậm/nhạt cho đỉnh đó và tiếp tục thao tác tìm kiếm trên đồ thị như bình thường. Cũng trong quá trình tìm kiếm, mỗi khi phát hiện thấy một cạnh nhạt nối hai đỉnh đậm, ta dừng lại ngay vì nếu gán nhãn tiếp sẽ gặp tình trạng một đỉnh có cả hai nhãn đậm/nhạt, trong trường hợp này, Blossom được phát hiện (xem tính chất của Blossom) và bị chập thành một đỉnh, thuật toán được bắt đầu lại với đồ thị mới cho tới khi trả lời được câu hỏi: "có tồn tại đường mở xuất phát từ x hay không?"

Nếu đường mở tìm được không đi qua đỉnh chập nào thì ta chỉ việc tăng cặp dọc theo đường mở. Nếu đường mở có đi qua một đỉnh chập thì ta lại nở đỉnh chập đó ra thành Blossom để thay đỉnh chập này trên đường mở bằng một đoạn đường xuyên qua Blossom:



Hình 89: Nở Blossom để dò đường xuyên qua Blossom

Lưu ý rằng không phải Blossom nào cũng bị chập, chỉ những Blossom ảnh hưởng tới quá trình tìm đường mở mới phải chập để đảm bảo rằng đường mở tìm được là đường đi cơ bản. Tuy nhiên việc cài đặt trực tiếp các phép chập Blossom và nở đỉnh khá rắc rối, đòi hỏi một chương trình với độ phức tạp $O(n^4)$.

Dưới đây ta sẽ trình bày một phương pháp cài đặt hiệu quả hơn với độ phức tạp $O(n^3)$, phương pháp này cài đặt không phức tạp, nhưng yêu cầu phải hiểu rất rõ bản chất thuật toán.

13.3. PHƯƠNG PHÁP LAWLER (1973)

Trong phương pháp Edmonds, sau khi chập mỗi Blossom thành một đỉnh thì đỉnh đó hoàn toàn lại có thể nằm trên một Blossom mới và bị chập tiếp. Phương pháp Lawler chỉ quan tâm đến đỉnh chập cuối cùng, đại diện cho Blossom ngoài nhất (Outermost Blossom), đỉnh chập cuối cùng này được định danh (đánh số) bằng đỉnh cơ sở của Blossom ngoài nhất.

Cũng chính vì thao tác chập/nở nói trên mà ta cần mở rộng khái niệm Blossom, có thể **coi một Blossom là một tập đỉnh nở ra từ một đỉnh chập** chứ không đơn thuần chỉ là một chu trình pha cơ bản nữa.

Xét một Blossom B có đỉnh cơ sở là đỉnh r. Với $\forall v \in B$, $v \neq r$, ta lưu lại hai đường pha từ r tới v, một đường kết thúc bằng cạnh đậm và một đường kết thúc bằng cạnh nhạt, như vậy có hai loại vết gãn cho mỗi đỉnh v (hai vết này được cập nhật trong quá trình tìm đường):

S[v] là đỉnh liền trước v trên đường pha kết thúc bằng cạnh đậm, nếu không tồn tại đường pha loại này thì S[v] = 0.

T[v] là đỉnh liền trước v trên đường pha kết thúc bằng cạnh nhạt, nếu không tồn tại đường pha loại này thì T[v] = 0.

Bên cạnh hai nhãn S và T, mỗi đỉnh v còn có thêm

Nhãn b[v] là đỉnh cơ sở của Blossom chứa v. Hai đỉnh u và v thuộc cùng một Blossom ⇔ b[u] = b[v].

Nhãn match[v] là đỉnh ghép với đỉnh v. Nếu v chưa ghép thì match[v] = 0.

Khi đó thuật toán tìm đường mở bắt đầu từ đỉnh x chưa ghép có thể phát biểu như sau:

Bước 1: (Init)

Hàng đợi Queue dùng để chứa những đỉnh đậm chờ duyệt, ban đầu chỉ gồm một đỉnh đậm x.

Với mọi đỉnh u, khởi gán b[u] = u và match[u] = 0 với $\forall u$.

Gán $S[x] \neq 0$; Với $\forall u \neq x$, gán S[u] = 0;Với $\forall v$: gán T[v] = 0

Bước 2: (BFS)

Lặp lại các bước sau cho tới khi hàng đợi rỗng:

Với mỗi đỉnh đậm u lấy ra từ Queue, xét những cạnh nhạt (u, v):

Nếu v chưa thăm:

Nếu v là đỉnh chưa ghép ⇒ Tìm thấy đường mở kết thúc ở v, dừng

Nếu v là đỉnh đã ghép ⇒ thăm v ⇒ thăm luôn match[v] và đẩy match[v] vào Queue.

Sau mỗi lần thăm, chú ý việc lưu vết (hai nhãn S và T)

Nếu v đã thăm

Nếu v là đỉnh nhạt hoặc b[v] = b[u] ⇒ bỏ qua

Nếu v là đỉnh đậm và b[v] ≠ b[u] ta phát hiện được blossom mới chứa u và v, khi đó:

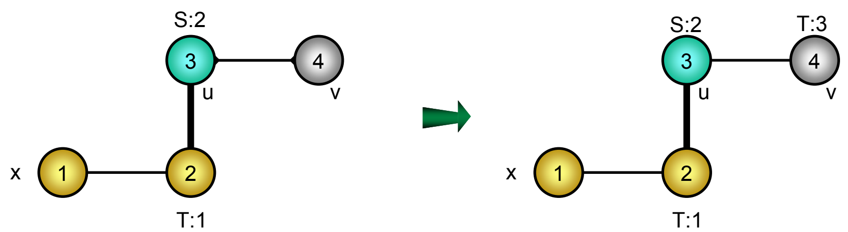
- **Phát hiện đỉnh cơ sở:** Truy vết đường đi ngược từ hai đỉnh đậm u và v theo hai đường pha về nút gốc, chọn lấy đỉnh a là đỉnh đậm chung gặp đầu tiên trong quá trình truy vết ngược. Khi đó Blossom mới phát hiện sẽ có đỉnh cơ sở là a.
- **Gán lại vết:** Gọi (a = i1, i2, …, ip = u) và (a = j1, j2, …, jq = v) lần lượt là hai đường pha dẫn từ a tới u và v. Khi đó (a = i1, i2, …, ip = u, jq = v, jq-1, …, j1 = a) là một chu trình pha đi từ a tới u và v rồi quay trở về a. Bằng cách đi dọc theo chu trình này theo hai hướng ngược nhau, ta có thể gán lại tất cả các nhãn S và T của những đỉnh trên chu trình. Lưu ý rằng **không được gán lại nhãn** S và T cho những đỉnh k mà b[k] = a, và với những đỉnh k có b[k] ≠ a thì **bắt buộc phải gán lại nhãn** S và T theo chu trình này bất kể S[k] và T[k] trước đó đã có hay chưa.

Chập Blossom: Xét những đỉnh v mà $b[v] \in \{b[i_1], b[i_2], \ldots, b[i_p], b[j_1], b[j_2], \ldots, b[j_q]\}$, gán lại b[v] = a. Nếu v là đỉnh đậm (có nhãn S[v] ≠ 0) mà chưa được duyệt tới (chưa bao giờ được đẩy vào Queue) thì đẩy v vào Queue chờ duyệt tiếp tại những bước sau.

Nếu quá trình này chỉ thoát khi hàng đợi rỗng thì tức là không tồn tại đường mở bắt đầu từ x.

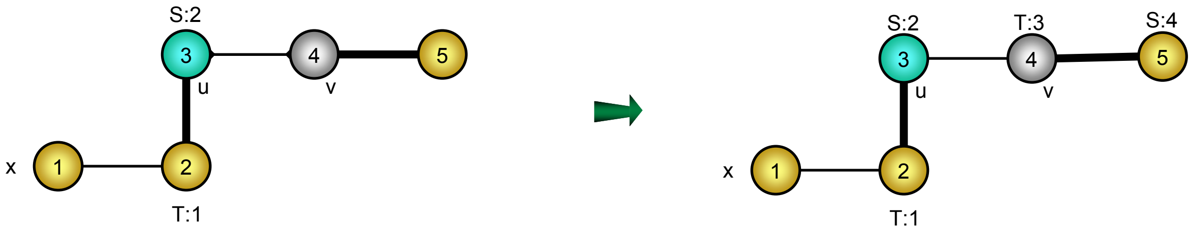
Sau đây là một số ví dụ về các trường hợp từ đỉnh đậm u xét cạnh nhạt (u, v):

Trường hợp 1: v chưa thăm và chưa ghép:



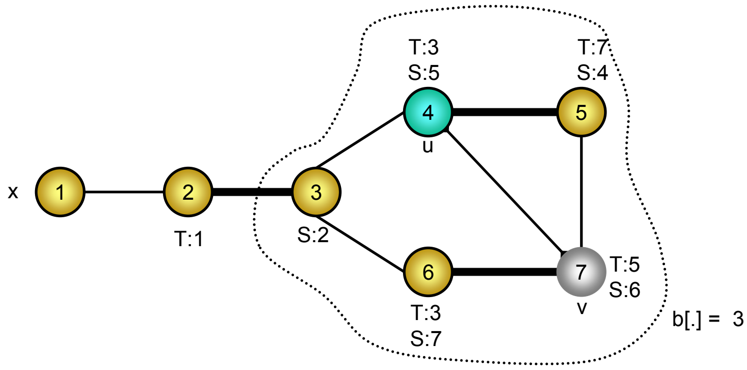
⇒ Tìm thấy đường mở

Trường hợp 2: v chưa thăm và đã ghép



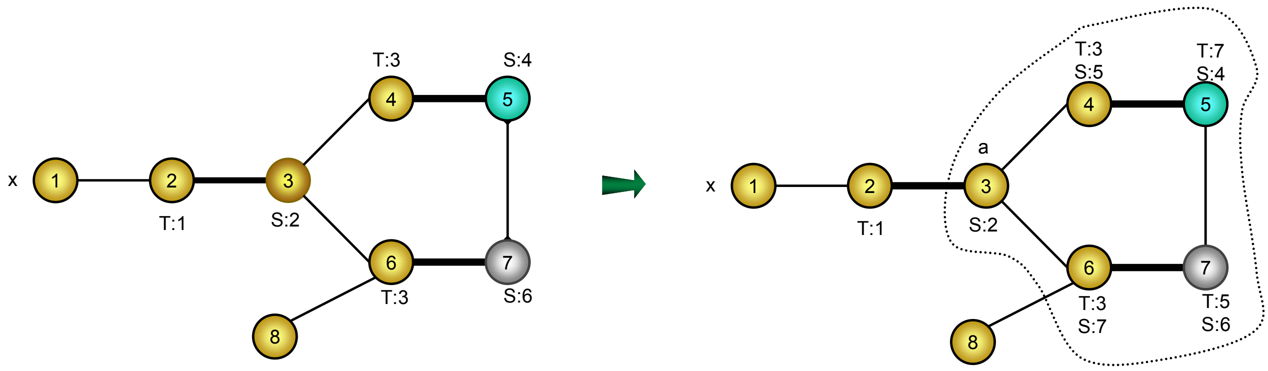
⇒ Thăm cả v lẫn match[v], gán nhãn T[v] và S[match[v]]

Trường hợp 3: v đã thăm, là đỉnh đậm thuộc cùng blossom với u



⇒ Không xét, bỏ qua

Trường hợp 4: v đã thăm, là đỉnh đậm và b[u] ≠ b[v]



⇒ Phát hiện Blossom, tìm đỉnh cơ sở a = 3, gán lại nhãn S và T dọc chu trình pha. Đẩy hai đỉnh đậm mới 4, 6 vào hàng đợi, Tại những bước sau, khi duyệt tới đỉnh 6, sẽ tìm thấy đường mở kết thúc ở 8, truy vết theo nhãn S và T tìm được đường (1, 2, 3, 4, 5, 7, 6, 8)

Tư tưởng chính của phương pháp Lawler là dùng các nhãn b[v] thay cho thao tác chập trực tiếp Blossom, dùng các nhãn S và T để truy vết tìm đường mở, tránh thao tác nở Blossom. Phương pháp này dựa trên một nhận xét: Mỗi khi tìm ra đường mở, nếu đường mở đó xuyên qua một Blossom ngoài nhất thì chắc chắn nó phải đi vào Blossom này từ nút cơ sở và thoát ra khỏi Blossom bằng một cạnh nhạt.

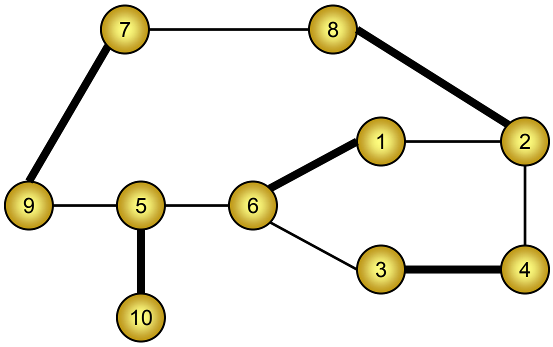
13.4. CÀI ĐẶT

Ta sẽ cài đặt phương pháp Lawler với khuôn dạng Input/Output như sau:

Input: file văn bản GMATCH.INP

- Dòng 1: Chứa hai số n, m lần lượt là số cạnh và số đỉnh của đồ thị cách nhau ít nhất một dấu cách ($n \le 100$)
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tượng trưng cho một cạnh (u, v) của đồ thị

Output: file văn bản GMATCH.OUT, ghi bộ ghép cực đại tìm được



GMATCH.INP	GMATCH.OUT
10 11	1) 1 6
1 2	2) 2 8
1 6	3) 3 4
2 4	4) 5 10
2 8	5) 7 9
3 4	
3 6	
5 6	
5 9	
5 10	
7 8	
7 9	

Chương trình này sửa đổi một chút mô hình cài đặt trên dựa vào nhận xét:

v là một đỉnh đậm ⇔ v = x hoặc match[v] là một đỉnh nhạt

Nếu v là đỉnh đậm thì S[v] = match[v]

Vậy thì ta không cần phải sử dụng riêng một mảng nhãn S[v], tại mỗi bước sửa vết, ta chỉ cần sửa nhãn vết T[v] mà thôi. Để kiểm tra một đỉnh $v \ne x$ có phải đỉnh đậm hay không, ta có thể kiểm tra bằng điều kiện: match[v] có là đỉnh nhạt hay không, hay T[match[v]] có khác 0 hay không.

Chương trình sử dụng các biến với vai trò như sau:

match[v] là đỉnh ghép với đỉnh v

b[v] là đỉnh cơ sở của Blossom chứa v

T[v] là đỉnh liền trước v trên đường pha từ đỉnh xuất phát tới v kết thúc bằng cạnh nhạt, T[v] = 0 nếu quá trình BFS chưa xét tới đỉnh nhạt v.

InQueue[v] là biến Boolean, InQueue[v] = True ⇔ v là đỉnh đậm đã được đẩy vào Queue để chờ duyệt.

start và finish: Nơi bắt đầu và kết thúc đường mở.

P_4_13_1.PAS * Phương pháp Lawler áp dụng cho thuật toán Edmonds

```
program MatchingInGeneralGraph;
const
  InputFile  = 'GMATCH.INP';
  OutputFile = 'GMATCH.OUT';
  max = 100;
var
  a: array[1..max, 1..max] of Boolean;
  match, Queue, b, T: array[1..max] of Integer;
  InQueue: array[1..max] of Boolean;
  n, first, last, start, finish: Integer;

procedure Enter;
var
```

```
  i, m, u, v: Integer;
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  FillChar(a, SizeOf(a), False);
  ReadLn(f, n, m);
  for i := 1 to m do
    begin
      ReadLn(f, u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  Close(f);
end;

procedure Init; {Khởi tạo bộ ghép rỗng}
begin
  FillChar(match, SizeOf(match), 0);
end;

procedure InitBFS; {Thủ tục này được gọi để khởi tạo trước khi tìm đường mở xuất phát từ start}
var
  i: Integer;
begin
  {Hàng đợi chỉ gồm một đỉnh đậm start}
  first := 1; last := 1;
  Queue[1] := start;
  FillChar(InQueue, SizeOf(InQueue), False);
  InQueue[start] := True;
  {Các nhãn T được khởi gán = 0}
  FillChar(T, SizeOF(T), 0);
  {Nút cơ sở của outermost blossom chứa i chính là i}
  for i := 1 to n do b[i] := i;
  finish := 0; {finish = 0 nghĩa là chưa tìm thấy đường mở}
end;

procedure Push(v: Integer); {Đẩy một đỉnh đậm v vào hàng đơi}
begin
  Inc(last);
  Queue[last] := v;
  InQueue[v] := True;
end;

function Pop: Integer; {Lấy một đỉnh đậm khỏi hàng đợi, trả về trong kết quả hàm}
begin
  Pop := Queue[first];
  Inc(first);
end;

{Khó nhất của phương pháp Lawler là thủ tục này: Thủ tục xử lý khi gặp cạnh nhạt nối hai đỉnh đậm p, q}
procedure BlossomShrink(p, q: Integer);
var
  i, NewBase: Integer;
  Mark: array[1..max] of Boolean;

  {Thủ tục tìm nút cơ sở bằng cách truy vết ngược theo đường pha từ p và q}
  function FindCommonAncestor(p, q: Integer): Integer;
  var
    InPath: array[1..max] of Boolean;
  begin
    FillChar(InPath, SizeOf(Inpath), False);
    repeat  {Truy vết từ p}
      p := b[p]; {Nhảy tới nút cơ sở của Blossom chứa p, phép nhảy này để tăng tốc độ truy vết}
      Inpath[p] := True; {Đánh dấu nút đó}
```

```
      if p = start then Break; {Nếu đã truy về đến nơi xuất phát thì dừng}
      p := T[match[p]]; {Nếu chưa về đến start thì truy lùi tiếp hai bước, theo cạnh đậm rồi theo cạnh nhạt}
    until False;
    repeat  {Truy vết từ q, tương tự như đối với p}
      q := b[q];
      if InPath[q] then Break; {Tuy nhiên nếu chạm vào đường pha của p thì dừng ngay}
      q := T[match[q]];
    until False;
    FindCommonAncestor := q; {Ghi nhận đỉnh cơ sở mới}
  end;

  procedure ResetTrace(x: Integer); {Gán lại nhãn vết dọc trên đường pha từ start tới x}
  var
    u, v: Integer;
  begin
    v := x;
    while b[v] <> NewBase do {Truy vết đường pha từ start tới đỉnh đậm x}
      begin
        u := match[v];
        Mark[b[v]] := True; {Đánh dấu nhãn blossom của các đỉnh trên đường đi}
        Mark[b[u]] := True;
        v := T[u];
        if b[v] <> NewBase then T[v] := u; {Chỉ đặt lại vết T[v] nếu b[v] không phải nút cơ sở mới}
      end;
  end;

begin {BlossomShrink}
  FillChar(Mark, SizeOf(Mark), False); {Tất cả các nhãn b[v] đều chưa bị đánh dấu}
  NewBase := FindCommonAncestor(p, q); {xác định nút cơ sở}
  {Gán lại nhãn}
  ResetTrace(p); ResetTrace(q);
  if b[p] <> NewBase then T[p] := q;
  if b[q] <> NewBase then T[q] := p;
  {Chập blossom ⇔ gán lại các nhãn b[i] nếu blossom b[i] bị đánh dấu}
  for i := 1 to n do
    if Mark[b[i]] then b[i] := NewBase;
  {Xét những đỉnh đậm i chưa được đưa vào Queue nằm trong Blossom mới, đẩy i và Queue để chờ duyệt tiếp tại các bước sau}
  for i := 1 to n do
    if not InQueue[i] and (b[i] = NewBase) then
      Push(i);
end;

{Thủ tục tìm đường mở}
procedure FindAugmentingPath;
var
  u, v: Integer;
begin
  InitBFS; {Khởi tạo}
  repeat  {BFS}
    u := Pop;
    {Xét những đỉnh v chưa duyệt, kề với u, không nằm cùng Blossom với u, dĩ nhiên T[v] = 0 thì (u, v) là cạnh nhạt rồi}
    for v := 1 to n do
      if (T[v] = 0) and (a[u, v]) and (b[u] <> b[v]) then
        begin
          if match[v] = 0 then {Nếu v chưa ghép thì ghi nhận đỉnh kết thúc đường mở và thoát ngay}
            begin
              T[v] := u;
              finish := v;
              Exit;
            end;
          {Nếu v là đỉnh đậm thì gán lại vết, chập Blossom …}
          if (v = start) or (T[match[v]] <> 0) then
            BlossomShrink(u, v)
          else {Nếu không thì ghi vết đường đi, thăm v, thăm luôn cả match[v] và đẩy tiếp match[v] vào Queue}
```

```
            begin
              T[v] := u;
              Push(match[v]);
            end;
        end;
  until first > last;
end;

procedure Enlarge; {Nới rộng bộ ghép bởi đường mở bắt đầu từ start, kết thúc ở finish}
var
  v, next: Integer;
begin
  repeat
    v := T[finish];
    next := match[v];
    match[v] := finish;
    match[finish] := v;
    finish := next;
  until finish = 0;
end;

procedure Solve; {Thuật toán Edmonds}
var
  u: Integer;
begin
  for u := 1 to n do
    if match[u] = 0 then
      begin
        start := u; {Với mỗi đỉnh chưa ghép start}
        FindAugmentingPath; {Tìm đường mở bắt đầu từ start}
        if finish <> 0 then Enlarge; {Nếu thấy thì nới rộng bộ ghép theo đường mở này}
      end;
end;

procedure Result; {In bộ ghép tìm được}
var
  u, count: Integer;
  f: Text;
begin
  Assign(f, OutputFile); Rewrite(f);
  count := 0;
  for u := 1 to n do
    if match[u] > u then {Vừa tránh sự trùng lặp (u, v) và (v, u), vừa loại những đỉnh không ghép được (match=0)}
      begin
        Inc(count);
        WriteLn(f, count, ') ', u, ' ', match[u]);
      end;
  Close(f);
end;

begin
  Enter;
  Init;
  Solve;
  Result;
end.
```

13.5. ĐỘ PHỨC TẠP TÍNH TOÁN

Thủ tục BlossomShrink có độ phức tạp O(n). Thủ tục FindAugmentingPath cần không quá n lần gọi thủ tục BlossomShrink, cộng thêm chi phí của thuật toán tìm kiếm theo chiều rộng, có độ phức tạp

$O(n^2)$. Phương pháp Lawler cần không quá n lần gọi thủ tục FindAugmentingPath nên có độ phức tạp tính toán là $O(n^3)$.

Cho đến nay, phương pháp tốt nhất để giải bài toán tìm bộ ghép tổng quát trên đồ thị được biết đến là của Micali và Varizani (1980), nó có độ phức tạp tính toán là $O(\sqrt{n}.m)$. Bạn có thể tham khảo trong các tài liệu khác.

TÀI LIỆU ĐỌC THÊM

[1] Christian Charras, Thierry Lecroq. ***Handbook of Exact String-Matching Algorithms.*** Gần 20 thuật toán tìm kiếm chuỗi, có diễn giải đầy đủ.

[2] Reinhard Diestel. ***Graph Theory***. Một cuốn sách chuyên về Lý thuyết đồ thị.

[3] Johan Håstad. ***Advanced Algorithms.***

[4] Andrew J. Manson. ***Speaker Matching.*** Bài báo nói về các thuật toán tìm bộ ghép trên đồ thị tổng quát, cả trong trường hợp đồ thị có trọng số

[5] Eva Milková. ***Graph Theory and Information Technology***. Một số thuật toán về bài toán cây bao trùm tối tiểu.

[6] Dave Mount. ***Design and Analysis of Computer Algorithms.***

[7] Nguyễn Xuân My, Trần Đỗ Hùng, Lê Sĩ Quang. ***Một số vấn đề chọn lọc trong tin học.*** Cuốn sách rất phù hợp cho học sinh phổ thông trung học yêu thích việc giải các bài toán tin học

[8] Nguyễn Đức Nghĩa, Nguyễn Tô Thành. ***Toán rời rạc***. Một cuốn sách rất căn bản dành cho sinh viên ngành tin học.

[9] Kenneth H. Rosen. ***Discrete Mathematics and its Applications (Bản dịch tiếng Việt: Toán học rời rạc ứng dụng trong tin học)***. Cuốn sách viết dưới dạng giáo trình rất dễ hiểu, có hệ thống bài tập được sắp xếp rất khoa học.

[10] Robert Sedgewick. ***Algorithms (Bản dịch tiếng Việt: Cẩm Nang Thuật Toán)***. Một cuốn sách rất tiện lợi cho tra cứu, đầy đủ các thuật toán kinh điển

黎明皇

In memory of committed teachers and excellent students.

Le Minh Hoang.